Interior Point Methods on GPU with application to Model Predictive Control

Gade-Nielsen, Nicolai Fog; Dammann, Bernd; Jørgensen, John Bagterp

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Interior Point Methods on GPU with application to Model Predictive Control

Nicolai Fog Gade-Nielsen

Kongens Lyngby 2014
PHD-2014-338

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 324, DK-2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk PHD-2014-338

Summary (English)

The goal of this thesis is to investigate the application of interior point methods to solve dynamical optimization problems, using a graphical processing unit (GPU) with a focus on problems arising in Model Predictice Control (MPC). Multi-core processors have been available for over ten years now, and many-core processors, such as GPUs, have also become a standard component in any consumer computer. The GPU offers faster floating point operations and higher memory bandwidth than the CPU, but requires algorithms to be redesigned and implemented, to match the underlying architecture.

A large number of different optimization algorithms are available for solving optimization problems. Some of the most common method are the simplex method and interior point methods. We focus on interior point methods in this thesis, due to its polynomial complexity, and since the use of the simplex method with GPUs have been investigated by several other authors already.

The main computational task in interior point methods is the solution of a linear system to compute the Newton direction in each iteration. Direct interior point methods use a direct method such as Cholesky factorization to factorize the normal equations of the Hessian matrix. The use of a GPU has been shown to be very efficient in the factorization of dense matrices, and several numeric libraries, which utilize the GPU, have become available during the course of this thesis. We have developed a direct interior point method, which utilizes the GPU, and demonstrate that our implementation can reduce the solution time substantially.

There are multiple software packages available for solving optimization problems

with interior point methods, such as GLPK, IPOPT, MOSEK and many more. However, none of these support the GPU yet. With this thesis, we include a new software package called GPUOPT, available under the non-restrictive MIT license. GPUOPT includes includes a primal-dual interior-point method, which supports both the CPU and the GPU. It is implemented as multiple components, where the matrix operations and solver for the Newton directions is separated from the core interior point method. This makes it possible to replace the matrix operations and solver with alternative, and potentially problem-specific, implementations.

In this thesis, we include different implementations of the matrix operations, including general dense, general sparse and problem-specific implementation of a test problem from model predictive control. Multiple solvers are implemented as well, including a direct solver based on CHOLMOD, and an iterative solver which uses preconditioned conjugate gradient. The iterative solver is based on the matrix-free iterative interior point method

Summary (Danish)

Målet for denne afhandling er at undersøge anvendelsen af indrepunktsmetoder til at løse dynamiske optimeringsproblemer ved brug af graphical processing unit (GPU), med fokus på problemer som opstår i model prædiktiv kontrol (MPC). Multi-core processorer har været til rådighed i over ti år nu, og many-core processorer, såsom GPU'er, er også blevet et standard komponent i computere. GPU'en byder på hurtigere beregninger og båndbredde end CPU'en, men kræver at algoritmer bliver omformuleret og implementeret, så de udnytter den underliggende arkitektur.

Mange forskellige optimeringsalgoritmer kan benyttes til at løse optimeringsproblemer. De mest gængse metoder er simpleks- og indrepunktsmetoderne. I denne afhandling fokuserer vi på indrepunktsmetoder, da den har polynomisk kompleksitet, mens simpleksmetoder har exponentiel kompleksitet. Desuden er brugen af en GPU til simpleksmetoder allerede blevet undersøgt af flere forfattere.

Den tungeste beregningsmæssige opgave i indrepunktsmetoder er løsningen af et lineært ligningssystem for at beregne Newton retningen i hver indrepunktsiteration. Direkte indrepunktsmetoder benytter sig af en direkte metode, såsom Cholesky faktorisering, til at løse ligningssystemet. Brugen af en GPU til Cholesky faktorisering af dense matricer er meget effektivt, og numeriske biblioteker, som benytter sig af en GPU til Cholesky faktorisering, er blevet udgivet i løbet af de sidste tre år. Vi har udviklet en direkte indrepunktsmetode, som benytter sig af GPU'en, og vist at vores implementering kan reducere beregningstiden væsenligt for dense systemer.

Der er mange forskellige software pakker til rådighed for at løse optimeringsprob-

lemer med indrepunktsmetoder, såsom GLPK, IPOPT, MOSEK, og mange flere. Ingen af disse pakker benytter sig på nuværende tidpunkt af GPU'en. Med denne afhandling udgiver vi en software pakke kaldet GPUOPT under en open-source licens. GPUOPT indeholder en implementering af en primal-dual indrepunktsmetode, som kan udnytte både en CPU og en GPU. Den er implementeret som komponenter, hvor matrixberegningerne og den linære løser er adskilt fra selve indrepunktsmetoden. Dette gør det muligt at erstatte implementeringen af matrixberegningerne og den linære løser med alternative, og eventuelt problemspecifikke, implementeringer.

I denne afhandling inkluderer vi forskellige implementeringer af matrixberegningerne for generelle dense og generelle sparse formatter, samt en problemspecifikke implementering for et test problem fra model prædiktiv kontrol. Flere forskellige lineære løsere er også implementeret, inklusiv en direkte løser baseret på CHOLMOD, og en iterativ løser som benytter sig af preconditioned conjugate gradient. Den iterative løser er baseret på en matrix-fri iterativ indrepunktsmetode.

Preface

This thesis was prepared at the Technical University of Denmark in fulfilment of the requirements for acquiring an Ph.D. degree. The work presented in this thesis was carried out during the period of November 2010 to April 2014 at the Department of Applied Mathematics and Computer Science, in the Scientific Computing section.

This project has been supervised by Associate Professors Bernd Dammann and John Bagterp Jørgensen. The work has been funded by grant no. 09-070032 from the Danish Research Council for Technology and Production Sciences (FTP).

Lyngby, April 2014

Nicolai Fog Gade-Nielsen

Acknowledgements

First, I would like to thank my main supervisor Assoc. Prof. Bernd Dammann and Assoc. Prof. John Bagterp Jørgensen for their guidance and advice during this project.

I would also like to thank the people involved in the GPUlab project at the Technical University of Denmark, including fellow Ph.D student Stefan L. Glimberg, Post.Doc. Hans Henrik B. Sørensen, Assoc. Prof. Allan P. Engsig-Karup and Prof. Per Christian Hansen for all the interesting discussions, helpful advice and inspiring conversations.

I would also like to thank Prof. Jacek Gondzio at the University of Edinburgh for my brief, but inspiring, stay which has greatly affected the direction of the second half of my project.

Finally, I would like to thank my family and friends for their endless support and patience.

Contents

Notation

The notations used throughout this thesis are listed below.

Symbol	Description
m	is the number of constraints in our optimization problem, which is equivalent to the number of rows in the constraint matrix A.
n	is the number of decision variables in our optimization problem, which is equivalent to the number of columns in the constraint matrix A.
N_t	is the number of time steps in the prediction horizon for our test case problem.
N_p	is the number of power plants for our test case problem.
N_u	is the number of control variables in our test case problem, which is equivalent to $N_t \times N_p$.
v_i	is the ith element of vector v.
$\|\|v\|\|_2$ or $\|\|v\|\|$	is the euclidean norm of vector v.
A^T	is the transpose of the matrix A.
x^T	is the transpose of the vector x.
$diag(d)$	is the diagonal matrix with the vector d as its diagonal elements.
$diag(A)$	is the vector with the elements of the diagonal of the matrix A
$x \times y$	is the multiplication of x times y.
e	is a vector of ones.
$(x, y) \geq 0$	means that all elements in the vector x and the vector y are greater than or equal to zero.

Glossary

The abbreviations used throughout this thesis are listed below.

Abbreviation	Description
BLAS	Basic Linear Algebra Subroutines.
CG	Conjugate Gradient.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
FIR	Finite Impulse Response.
GPU	Graphics Processing Unit.
IPM	Interior Point Method.
KKT	Karush-Kuhn-Tucker.
LAPACK	Linear Algebra PACKage.
MCC	Multiple Centrality Correctors.
MISO	Multiple-Input Single-Output.
MPC	Model Predictive Control.
PCG	Preconditioned Conjugate Gradient.
SIMT	Single-Instruction Multiple-Thread.
WCD	Weighted Corrector Directions.

Chapter 1

Introduction

1.1 Motivation

The application of Model Predictive Control (MPC) and dynamical optimization in general is computationally challenging due to the large number of variables, as well as the constraints imposed by the real-time requirement.

The Graphical Processing Unit (GPU) is the processing chip on modern graphic cards. Due of the ever-increasing demand for advanced graphics, the GPU has developed into a massively parallel chip with very fast floating point computation and high memory bandwidth. These properties makes the GPU attractive for not only graphics rendering, but also many different scientific computations. GPU computing has shown itself to be extremely efficient at certain numerical computations. GPUs provide a significant increase in computing performance and memory bandwidth compared to traditional CPUs (Central Processing Units). The cost of these benefits is a more complicated programming model and certain restrictions on the type of workload which can fully utilize the performance. Not all numerical computations are equally suited for the architecture, due to the massive parallelism and restrictions on how memory can be accessed to achieve optimal performance.

1.2 Objective and main contribution

The main objective of this work is to implement and evaluate the use of many-core GPUs for solving constrained optimization problems arising in dynamical optimization, such as Model Predictive Control. These problems are usually solved using either a simplex method or an interior point method. In this project, we have limited ourselves to the primal-dual interior point method. The use of the simplex method with GPUs has been looked at by many authors [SE09, BPM10, LBEB11, MAC11], while interior point methods which utilize GPUs are still rare. One example of such is in [HJO07], which utilizes the GPU for Cholesky factorization. Another work with interior point methods on the GPU is [SGH12], where the GPU is used for sparse matrix-vector multiplication in an interior point method with an iterative solver.

Our first goal in this thesis is to determine the benefit of using a GPU to accelerate the interior point method presented in [ESJ09]. The method presented in [ESJ09] exploits the structure of a constrained optimization problem from model predictive control (MPC) and shows a speed-up of an order-of-magnitude compared to a generic solver. To determine whether a GPU can help accelerate this method, we implement their method in MATLAB, extend the implementation with the built-in GPU support in MATLAB, and then implement the algorithm entirely in C with CUDA.

Our second goal in this thesis is to evaluate the matrix-free interior point method from [Gon12b] with our model predictive control problem. This is done by implementing the method described in [Gon12b] and extending the partial Cholesky factorization as well as the conjugate gradient implementation to use a GPU.

Our third goal is to combine the implemented methods into a toolbox with an user-friendly interface and release it as open source software.

1.3 Outline

Chapter 2 introduces the theory relevant to this thesis. It introduces the programming model, which is used to program many-core GPUs from NVIDIA, as well as model predictive control and primal-dual interior point methods with focus on linear optimization problems.

Chapter 3 describes our test case, which is a power plant portfolio system from model predictive control. The transfer function of the power plants is described and converted into discrete state space form and the test case is defined as an linear optimization problem in the inequality form and the standard form.

Chapter 4 presents a primal-dual interior point method for linear optimization problems in inequality form based on [ESJ09] and describes the implementation from MATLAB to full GPU implementation. The interior point method is further specialized for the test-case from Chapter 3, where the structure of the model predictive control problem is exploited to reduce the solution time.

Chapter 5 presents a newly developed modular optimization toolbox called GPUOPT. It contains a primal-dual interior point method for linear optimization problems in the standard form and the inequality form based on Mehrotra predictor-corrector algorithm [Meh92]. The algorithm is implemented on both the CPU and the GPU and features primal-dual regularization [AG99], multiple centrality correctors [Gon96] and weighted corrector directions [CG08]. The toolbox is implemented to be modular, such that matrix operations and linear solving is separated from the core interior point method implementation. This allows for multiple different implementations of the matrix operations and the linear solver.

Chapter 6 presents an implementation of the matrix-free preconditioned conjugate gradient solver [Gon12b] for GPUOPT. The implementation is done for both CPU and GPU and demonstrates that the use of GPUs can substantially accelerate the solution time of an interior point method when using iterative methods to solve the linear system of equations to compute the Newton search directions.

Chapter 7 summarizes the results and main contributions of this thesis.

Appendix A lists papers published during this thesis.

1.4 Hardware and software used for testing

All the tests in this thesis have been executed on a machine at GPUlab at DTU Compute. Table 1.1 lists the specifications of the machine.

Table 1.1: Technical specifications for test machine

Name	GPUlab06
CPU	Intel(R) Core(TM) i7-3820 3.6 GHz
CPU memory	32 GB DDR3-1333
CPU bandwidth	42 GB/s
GPU	2× NVIDIA Tesla K-20C 5 GB
GPU memory	5 GB
GPU bandwidth	208 GB/s
Software	Ubuntu 10.04 LTS ATLAS 3.10.1, MAGMA 1.4.1, CUDA 5.5, MATLAB R2013b, SuiteSparse 4.2.1

Chapter 2

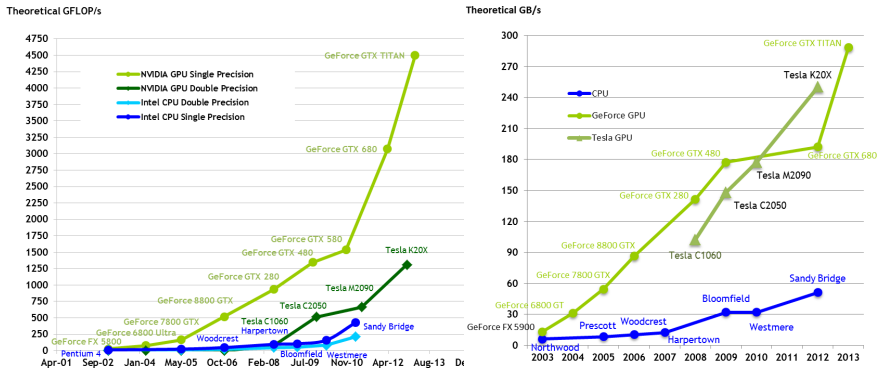
Theory

In this chapter, we introduce the different theoretical concepts used in the thesis. This includes a basic introduction to model predictive control, GPU computing, and interior-point methods.

2.1 GPU computing

The Graphical Processing Unit (GPU) is the main processor on a graphics card. Driven by increasing demands of the gaming and graphics industry, the GPU has developed into a massively parallel processor with very high floating-point performance and memory bandwidth. Figure 2.1 illustrates the theoretical performance of the many-core GPUs from NVIDIA compared to the multi-core CPUs from Intel.

Multi-core CPUs have a few large cores with large caches, which are designed for fast sequential performance to achieve minimum latency. Instead of minimizing latency, the GPU relies on massive parallelism to achieve maximum throughput and thus hide the latency of memory accesses. This is why there is such a large gap between the performance of the CPU and the GPU.



(a) Floating point operations per second **(b)** Memory bandwidth

Figure 2.1: Theoretical peak performance for NVIDIA GPUs and Intel CPUs. Taken from [NVI13b].

While the GPU has a clear advantage over the CPU when it comes to theoretical performance, the practical performance is highly problem dependent. Some operations, such as dense matrix-matrix multiplication, which is inherently parallel and structured, benefits greatly from massive parallelism and high memory bandwidth of the GPU. The performance benefit for other operations, such as inherently sequential operations like sparse Cholesky factorization, is less obvious as only sub-operations can be parallelized [VCC+10].

This has resulted in extensive research in a broad range of scientific areas to determine whether the GPU is applicable to accelerate computations in various fields. Algorithms which are optimal for the CPU are not necessarily optimal for the GPU. A key part of scientific GPU computing research is to determine the algorithms, which suit the architecture of the GPU, and implement them in an efficient manner.

2.1.1 Programming GPUs

Modern GPUs are designed to be programmable and frameworks have been designed to simplify the programming of the GPU. The two most widespread programming models for GPUs are CUDA and OpenCL.

CUDA is the leading proprietary programming model for GPUs. It is developed by NVIDIA and only supports NVIDIA GPUs. It is the most mature programming model available with many open-source and proprietary libraries

available.

OpenCL was originally developed by Apple, but is now managed by the Khronos Group. Unlike CUDA, OpenCL is a cross-platform framework, such that it can be used on both NVIDIA GPUs and AMD GPUs, as well as many other parallel architectures including CPUs, FPGAS and the Intel MIC architecture. The only limitation is that each vendor must implement OpenCL support for their particular architecture. However, while OpenCL supports multiple different architecture and the same code should theoretically work across platforms, it must be optimized for each of the targeted architectures to achieve optimal performance.

In this work, we restrict ourselves to the CUDA programming model due to its large availability of mature libraries and tools. However we would like to see OpenCL implementations of our code in the future as it an open standard and its performance is similar to CUDA [FVS11].

In the following sections, we introduce the CUDA architecture and programming model. This serves as an quick introduction to GPU computing with CUDA and its challenges, as an basic understanding of these topics is necessary to understand the implementation challenges in the following chapters. For a more thorough introduction and additional details, we refer to the excellent book [KmWH10] and the NVIDIA documentation [NVI13b, NVI13a].

2.1.2 CUDA physical architecture

A CUDA-capable GPU is normally located on a discrete graphics card connected to the host through the PCI-express bus, as illustrated in Figure 2.2 on the following page. The GPU is connected to an off-chip memory on the graphics card, called the device memory, which is typically between 1 GB to 6 GB, with the current state-of-the-art being the NVIDIA Tesla K40 with 12 GB of device memory.

The GPU can read and write to the device memory over the local device bus with a theoretical bandwidth ranging between 150 GB/s to 300 GB/s, depending on the graphics card, as shown in Figure 2.1b. It is often not possible to reach the theoretical bandwidth in practice. The achievable bandwidth is usually around 75% of the theoretical bandwidth.

The host can copy data between the host memory and device memory over the PCI-express bus. The theoretical bandwidth of the PCI-express bus is up to 8 GB/s for PCI-express 2.0 and 16 GB/s for PCI-express 3.0.

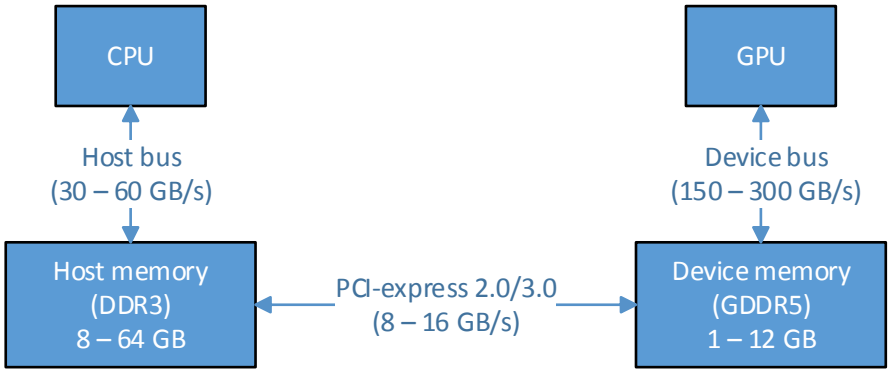


Figure 2.2: Diagram showing the interconnect between the CPU (host) and the GPU (device). The specified memory bandwidth and memory sizes are based on what is generally available in desktop machines at the time of writing.

Due to the much smaller bandwidth of the PCI-express bus compared to the device bus, it should be avoided to transfer data between the host and the device as much as possible.

2.1.2.1 Basic architecture

Figure 2.3 shows a basic overview of the architecture of a CUDA-capable GPU. At the time of writing, NVIDIA has released multiple different CUDA architectures, such as the Fermi [NVI09] and Kepler architecture [NVI12], but the basic architecture is the same.

The GPU consists of a number of streaming multiprocessors (SMs) and usually also a small L2 cache for the device memory. Each SM contains of a number of CUDA cores, registers, and a small memory, called shared memory. For the Fermi architecture and newer architectures, an L1 cache was added to the SM. The shared memory is located in the same memory as the L1 cache. GPUs based on the Kepler architecture also give the programmer direct access to an additional 48 KB read-only cache, called the texture cache, which was originally only available to the programmer by mapping textures.

The number of SMs in an GPU depends on the model and the number of cores per SM depends on the GPU architecture. For instance, the NVIDIA Tesla K20 installed in our test machine, described in Section 1.4 on page 4, is a Kepler

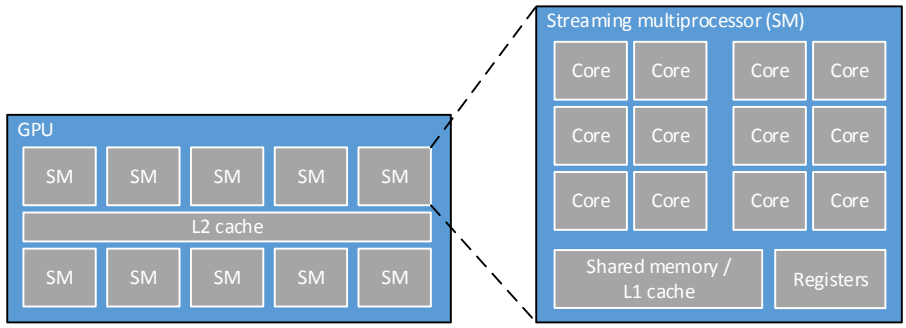


Figure 2.3: Simplified diagram of the architecture of a CUDA-capable GPU.

GPU and contains 13 SMs with 192 cores per SM for a total of 2496 cores.

2.1.2.2 Compute capability

All CUDA-capable GPUs have a compute capability version, currently ranging from 1.0 to 3.5. The compute capability defines various properties for the GPU, such as the number of registers in an SM and the size of shared memory. For more details, see [NVI13b].

2.1.2.3 Single-Instruction Multiple-Thread

The threads running on the CUDA cores are executed in a Single-Instruction Multiple-Thread (SIMT) manner. Each CUDA core executes a single thread, but does not have its own scheduler. Instead, the threads are divided into warps and the SM contains a number of warp schedulers. All the threads in the same warp execute the same instruction, but with their own data.

The SIMT architecture facilitates branching within a warp by splitting threads into active threads and inactive threads. When there is a branch in the code, which splits the execution path of threads within a warp, the warp scheduler executes each of the branches sequentially. This is done by setting the threads evaluating the first branch to active and the other threads in the warp to inactive, and then executing the first branch for only the active threads. Once the first branch has been executed, the second branch is executed for the remaining threads in the same manner. This results in an execution time which is the sum of executing each branch and should be avoided when possible.

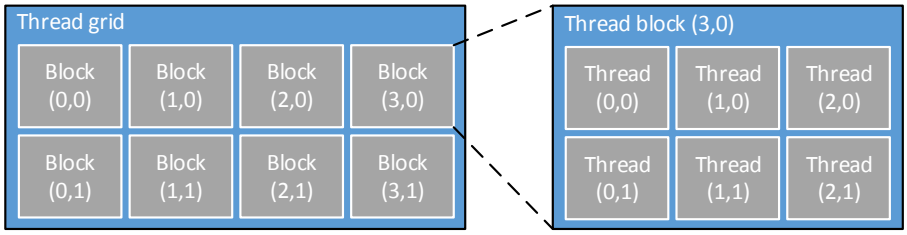


Figure 2.4: Thread grid and thread blocks in CUDA programming model.

The number of threads in a warp is defined by the compute capability of the GPU. At the time of writing, a warp is defined as 32 threads for all the compute capability versions up to 3.5.

2.1.3 CUDA programming Model

CUDA provides an abstract scalable programming model. It is designed to be an extension to C and CUDA also supports a subset of C++, such as templates. The GPU is programmed by implementing device functions, called kernels. A kernel is implemented like a standard function, however it is prefixed the `__global__` declaration to indicate it is a device function. This informs the CUDA compiler to generate device code instead of host code, when compiling the source file.

2.1.3.1 Kernel execution and threads

When a kernel is called, a thread grid is created to execute the kernel on the GPU. A thread grid is a three dimensional grid of thread blocks and each thread block is a three dimensional grid of CUDA threads. The dimensions of the thread blocks and the thread grid are specified by the programmer, which results in the number of threads created. For instance, if the programmer runs the kernel with a thread grid of dimensions $(10, 5, 1)$ and with thread blocks of dimensions $(16, 16, 1)$, then 50 thread blocks are created with each 256 threads. This results in 12800 threads, which execute the kernel. Figure 2.4 illustrates an 2D example of a thread grid of dimensions $(4, 2, 1)$ with thread blocks of dimension $(3, 2, 1)$.

The compiler determines the required number of registers per thread and shared memory per thread block when compiling a kernel. When the kernel is executed, the thread blocks in a thread grid are distributed to the SMs on the GPU. An

SM can contain multiple thread blocks, depending on the amount of resources a thread block uses and the compute capability of the GPU.

Threads within the same thread block are able to synchronize execution and share data by using the shared memory in an SM, but there is no synchronization available between different thread blocks. This allows for scalability, as the CUDA scheduler can run the thread blocks in any order and in parallel.

All the threads in a thread grid execute the exact same kernel. Each thread can use its indices in the thread block and the indices of the thread block in the thread grid to control its behavior. A simple example would be an array copy kernel, which copies the elements of an input array to an output array. A thread would use its indices to compute a unique thread index corresponding to an index in the array. It would then load the corresponding element from the input array and store the element in the same location in the output array.

2.1.3.2 Global memory

The device memory is the large off-chip memory, which is split into mostly global memory and a small amount of local memory. The global memory is visible to all the threads on the GPU. While the bandwidth of global memory is much higher than the memory bandwidth of a CPU, the access latency is much larger. CPUs use large caches to hide the memory latencies, however the GPU uses parallelism, although some small caches are now available in the GPU to assist as well. When threads in a warp access global memory, the GPU switches execution to other warps resident on the SM while waiting for memory operation to complete.

Another aspect of global memory is memory alignment and coalescence. Reads and writes to global memory are done as 32-, 64-, or 128-bytes memory transactions, aligned to memory locations, which are a multiple of the transaction size. The GPU coalesces memory accesses by a warp into as few memory transactions as possible to minimize the memory bandwidth used. For optimal memory performance, it is necessary to implement the kernel such that threads within the same warp read memory locations within the same memory segment.

2.1.3.3 Shared memory

As mentioned above, a SM contains a small memory called shared memory. The thread blocks can allocate memory from shared memory and use it to share data

between the threads in the thread block. Since shared memory is located inside the SM, the access latency of shared memory is around the same latency as the registers and thus much less than global memory.

Utilizing shared memory is key to improving performance of memory-bound kernels, which are kernels where the performance is limited by the memory bandwidth. For instance, shared memory can be used to achieve memory coalescence or to handle cooperation between threads in a thread block. An obvious example would be a reduction operation, such as sum reduction. Each thread in a thread block can load different values in global memory, store the values in shared memory, reduce the value to a single value in shared memory and then write the result to global memory. We will describe an example of a reduction kernel in Section 4.2.3.3.

2.1.4 CUDA libraries

There are many different CUDA libraries available. Utilizing libraries reduces the workload required to implement efficient code on the GPU and also benefits from performance improvements whenever the library is updated. In this section, we briefly introduce the CUDA libraries used in the thesis. We have restricted the libraries to only freely available libraries.

CUBLAS [NVIa] is the BLAS equivalent library for CUDA provided by NVIDIA. It contains all the standard BLAS functions for dense vector and matrix operations.

CUSPARSE [NVIb] is the sparse matrix library, also provided by NVIDIA. It contains various functions for sparse vector and sparse matrix operations in multiple different sparse formats.

MAGMA [ICL] is a dense linear algebra library, similar to LAPACK, for heterogeneous architectures, including GPU support. It contains a very fast implementation of Cholesky factorization on NVIDIA GPUs [LTND10]. It is developed by the Innovative Computing Laboratory at the University of Tennessee.

Thrust [NVId] is a data manipulation library for CUDA written in C++, which supports operations such as vector sorting. It is provided by NVIDIA and also includes an CPU implementation of all the operations.

These are just the libraries used in this thesis. A more complete list of libraries for CUDA can be found in [NVIc].

2.2 Model predictive control

In this section, we introduce the basics of model predictive control (MPC), as used in this thesis. For more details, we refer to [Mac02, CB04, RM09].

Model Predictive Control (MPC) is a method of process control, which computes the optimal control input for a dynamical system, by using a model of the system to predict the future states and outputs.

Linear dynamical system models can be formulated in many different ways, such as an impulse response model, a step response model, or a transfer function model. All of these models can be converted to a discrete time state space model [HJPM14], which has the following form,

$$x_{k+1} = Ax_k + Bu_k \tag{2.1}$$
$$y_k = Cx_k + Du_k \tag{2.2}$$

where x_k is the state, u_k is the input, y_k is the output of the system. Based on the state and the input at the discrete time step k, the matrices A and B define the linear model to compute the next state of the system, and the matrices C and D define the model to compute the output of the system.

The control objective for an MPC problem is usually defined as an ℓ_2-penalty function, an ℓ_1-penalty function, or an economic penalty function [Edl10]. The objective may penalize deviation from a set-point trajectory or aim to maximize profits. Using an ℓ_1-penalty function, or an economic penalty function as control objective, results in a linear optimization problem, while an ℓ_2-penalty function results in a quadratic optimization problem. In this thesis, we focus on the economic penalty function, which has the form

$$\phi_{\text{eco}} = \sum_{k=0}^{N-1} c_{u,k}^T u_k \tag{2.3}$$

where $c_{u,k}$ is the linear cost for u_k.

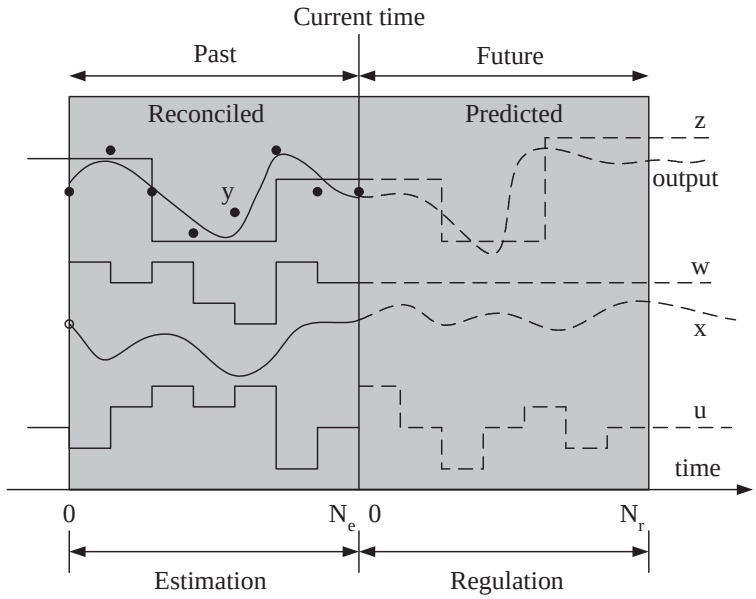
The system model and control objectives are formulated as a constrained optimization problem, where the control objective is defined as the penalty function and the model is defined as equality constraints:

$$\begin{aligned} \min \quad & \phi && && (2.4a) \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k && k = 0, 1, \ldots, N_t - 1 && (2.4b) \\ & y_k = Cx_k + Du_k && k = 1, 2, \ldots, N_t && (2.4c) \end{aligned}$$

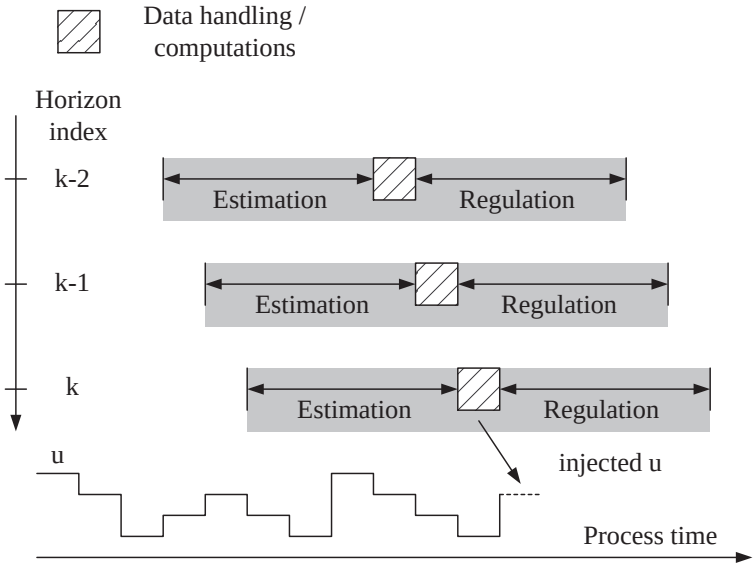
where N_t is the number of discrete time steps in the prediction horizon. The prediction horizon defines how many future time steps the controller considers. Model predictive control problems generally also have some kind of constraints on the input, such as bounds on the minimum and maximum value of the input, and constraints on the output, such as a set-point trajectory. One of the advantages of MPC is that it easily handles additional constraints by simply adding them to the optimization problem. We demonstrate this for our test problem in Chapter 3.

In each time step, the controller solves the optimization problem and computes a control trajectory for the next N_t time steps, based on the current state of the system, the model, and a reference trajectory. Only the control inputs for the current time step are actuated on the system. In the next time step, the controller uses past measurements to estimate the new state and solves the optimization problem again, to compute a new control trajectory. This strategy is called receding horizon control, and is shown on Figure 2.5. By continuously estimating the state and recomputing a new control trajectory in each time step, MPC can handle unknown disturbances and model uncertainties.

The requirement of solving the optimization problem in each time step results in a real-time constraint, as the solution must be computed in the interval between the current time step and the next time step. Computing the solution to the optimization problem in real-time can be computationally intensive, depending on the size and the time scale of the problem. This is one of the motivations for utilizing GPUs to reduce the solution time.



(a) Estimation and regulation in model predictive control.



(b) Online computation and moving horizons.

Figure 2.5: Moving horizon estimation and control.

2.3 Interior point method

Interior point methods are a class of algorithms which can be used to solve mathematical optimization problems. There are many different kinds of interior point methods for different type of problems. In this work, we focus on the primal-dual long-step path-following interior point method for linear optimization problems. In the following sections, we introduce this method along with practical information such as computing an initial point which is required for an efficient practical implementation of the method.

Our presentation is based on [NW06] and [CG08], which includes excellent introductions to practical interior point methods. For more information on interior point methods, including theoretical proofs and practical application, we refer to [NW06, CG08, Gon12a, AGMX96, Meh92] and the references therein.

2.3.1 Basics

A linear optimization problem in standard form is written as

$$\begin{array}{lll} & \text{Primal} & \\ \min\limits_{x} & \phi_p = c^T x & \\ \text{s.t.} & Ax = b & \\ & x \geq 0 & \end{array} \qquad \begin{array}{lll} & \text{Dual} & \\ \max\limits_{y,s} & \phi_d = b^T y & \\ \text{s.t.} & A^T y + s = c & \\ & s \geq 0 & \end{array} \tag{2.5}$$

where c is the linear objective, A is the constraints matrix, x is the primal solution, y is the dual solution and s is the dual slack.

The first order optimality conditions, also known as the Karush-Kuhn-Tucker (KKT) conditions, for the optimization problem can be written as

$$c - A^T y - s = 0 \tag{2.6a}$$
$$b - Ax = 0 \tag{2.6b}$$
$$XSe = 0 \tag{2.6c}$$
$$(x, s) \geq 0 \tag{2.6d}$$

where X and S are diagonal matrices with x and s as their diagonal, respectively, and e is a vector of ones. The optimal solution to (2.5) can be determined by computing the solution to the KKT conditions in (2.6).

The equality constraints (2.6a) to (2.6c) can be written in matrix form as

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} x \\ y \\ s \end{bmatrix} = \begin{bmatrix} c \\ b \\ 0 \end{bmatrix} \tag{2.7}$$

Primal-dual interior point methods use Newton's method to solve the equality constraints by linearizing around the current iterate, (x, y, s), to obtain the search direction, $(\Delta x, \Delta y, \Delta s)$ by solving the linear system

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} c - A^T y - s \\ b - Ax \\ -XSe \end{bmatrix} = \begin{bmatrix} r_d \\ r_p \\ r_c \end{bmatrix} \tag{2.8}$$

The search direction for this system is called the pure Newton direction, also known as the affine scaling direction. As the Newton method cannot handle the inequality constraint (2.6d), it is necessary to perform a line search to determine a step length, α, such that the next iterate

$$(x, y, s) = (x, y, s) + \alpha(\Delta x, \Delta y, \Delta s) \tag{2.9}$$

does not violate the inequality constraint. Unfortunately, the affine search direction usually does not make much progress towards the solution, as the step length is usually very small before violating the inequality constraint [NW06].

2.3.2 Primal-dual path-following interior point method

The primal-dual feasible region can be defined as

$$\mathcal{F} = \{(x, y, s) : Ax = b, A^T y + s = c, (x, s) \geq 0\} \tag{2.10}$$

and the primal-dual strictly feasible region is defined as

$$\mathcal{F}^0 = \{(x, y, s) : Ax = b, A^T y + s = c, (x, s) > 0\} \tag{2.11}$$

The set $\mathcal{F}^0$ is also known as the interior of the feasible region.

To improve the step length of the search direction, primal-dual path-following interior point methods do not compute the search direction by directly solving the affine scaling direction. Instead, they define a path inside the interior of the feasible region, leading to the optimal point and compute the Newton directions by solving a linear system aiming at a point on this path instead of directly at the optimal point.

We restrict ourselves to the path defined by the logarithmic barrier function. The logarithmic barrier problem of the primal in the linear optimization problem in (2.5) is written as

$$\begin{aligned} \min_{x} \quad & \phi_p = c^T x - \mu \sum_{i=1}^{n} \ln(x_i) \\ s.t. \quad & Ax = b \\ & x > 0 \end{aligned} \tag{2.12}$$

In this problem, the added logarithmic term in the objective function penalises iterates too close to the boundary. This forces the iterates to stay in the interior of the feasible region. The penalty can be controlled with the parameter μ, which is known as the barrier parameter or barrier term.

The Lagrangian for the barrier problem is written as

$$L(x, y) = c^T x - \mu \sum_{i=1}^{n} \ln(x_i) - y^T (Ax - b) \tag{2.13}$$

which gives us the perturbed KKT conditions,

$$\begin{aligned} \nabla_x L(x, y) = c - \mu X^{-1} e - A^T y = 0 \\ \nabla_y L(x, y) = b - Ax = 0 \\ x > 0 \end{aligned} \tag{2.14}$$

By introducing $s = \mu X^{-1} e$, we express the perturbed KKT conditions as

$$\begin{aligned} A^T y + s = c \\ Ax = b \\ XSe = \mu e \\ (x, s) > 0 \end{aligned} \tag{2.15}$$

The solutions to (2.15) for $\mu > 0$ defines a continuous smooth curve in the interior of the feasible region leading to the optimal solution as μ goes towards zero. This curve is called the primal-dual central path. Instead of computing the affine Newton direction, the primal-dual path-following interior point method computes the Newton direction for the perturbed KKT conditions which are

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} r_d \\ r_p \\ r_c + \sigma \mu e \end{bmatrix} \tag{2.16}$$

where $\sigma \in (0, 1)$ is called the centering parameter. For $\sigma = 1$, the Newton direction is called a centering direction which aims towards the point on the

central path for the value of μ. For $\sigma = 0$, the Newton direction is called the affine direction as it is equivalent to the Newton direction in (2.8).

The barrier parameter, μ, is set to

$$\mu = \frac{x^T s}{n} \tag{2.17}$$

where n is the number of elements in x and s. This value is known as the duality measure and for a given iterate, it provides an estimate of optimality, which indicate how close we are to the optimal solution.

Given this value of μ in (2.17), the centering step aims only to increase the centrality of the current iterate. That is, it attempts to bring the iterate closer to the central path, but does not aim to bring the iterate closer to optimality. As this step is biased towards the interior of the feasible region, it is usually possible to take longer steps than with the affine Newton direction. In contrast, the affine step aims at reducing the optimality in one step, but ignores the central path, often resulting in a small step length, due to hitting the boundary of the interior feasible region.

The primal-dual path-following algorithm computes the Newton direction by solving (2.16) with a σ in the open interval $(0, 1)$, which attempts to both reduce μ while maintaining centrality to avoid small step length.

The algorithm restricts the iterates within a neighborhood of the central path. Two common neighborhoods around the central path are the tight neighborhood

$$\mathcal{N}_2(\theta) = \{(x, y, s) \in \mathcal{F}^0 : \|XSe - \mu e\|_2 \leq \theta\mu\} \tag{2.18}$$

and the wide neighborhood

$$\mathcal{N}_{-\infty}(\gamma) = \{(x, y, s) \in \mathcal{F}^0 : x_i s_i \geq \gamma\mu,\ i = 1, 2, \ldots, n\} \tag{2.19}$$

The $\mathcal{N}_2(\theta)$ neighborhood is a tight region around the central path and algorithms based on this neighborhood are called short-step path-following algorithms. The $\mathcal{N}_{-\infty}(\gamma)$ is a much larger region around the central path and a typical value of γ is 10^{-3}, which results in a neighborhood that contains most of the feasible region [NW06]. This neighborhood is illustrated on Figure 2.6. Algorithms based on this neighborhood are called long-step path-following algorithms.

Short-step path-following algorithms are of theoretical importance, as they maintain the best theoretical convergence result. In practical implementations, the long-step algorithm is preferred as the number of iterations to converge in the short-step algorithm often approaches the worst-case. In this thesis, we restrict ourselves to the long-step algorithm as we focus on practical implementation.

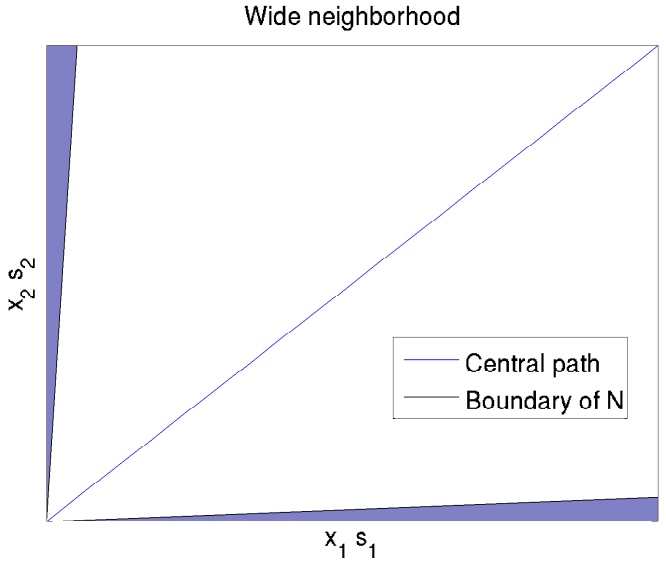


Figure 2.6: Example of the wide neighborhood, $\mathcal{N}_{-\infty}(\gamma)$, around the central path with $\gamma = 0.05$, indicated by the white region, for a problem with two variables.

Listing 2.1 summarizes the primal-dual long-step path-following interior point method. The maximum step length is computed for the Newton direction, which allows the complementarity pair (x, s) to remain positive. When the step is taken, the scaling parameter $\eta \in (0, 1)$ is used to ensure that the pair remains strictly positive. Typically η has a value of 0.9995 or 0.99 [NW06].

Interior point theory focuses on a good choice of σ to balance the trade-off between increasing centrality and making good progress towards the optimal solution. In the following section, we will describe a popular method for choosing σ and the search direction, which forms the basis for our implementations.

2.3.3 Mehrotra's predictor-corrector algorithm

Mehrotra's predictor-corrector algorithm [Meh92] splits the computation of the search direction in (2.16) into two parts by first computing a predictor direction, which aims at improving optimality, and then computing a corrector direction, which aims at maintaining centrality. The predictor-corrector algorithm at-

Listing 2.1: Primal-dual long-step path-following interior point method

Given (x_0, y_0, s_0) and η
for $k = 0, 1, 2, \ldots$ and not converged
  Choose $\sigma_k \in (0, 1)$
  Compute $(\Delta x_k, \Delta y_k, \Delta s_k)$ by solving (2.16)
  Compute step length α_k such that
    $x_k + \alpha_k \Delta x_k > 0$ and $s_k + \alpha_k \Delta s_k > 0$.
  Take step
    $(x_{k+1}, y_{k+1}, s_{k+1}) = (x_k, y_k, s_k) + \eta \alpha_k (\Delta x_k, \Delta y_k, \Delta s_k)$
end

tempts to compensate for the linearization error in the affine-scaling direction [NW06].

The predictor direction is computed by solving the affine-scaling direction in (2.8). The computed search direction, $(\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta s^{\text{aff}})$ is then used to estimate the linearization error and a second order corrector is computed by solving

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cor}} \\ \Delta y^{\text{cor}} \\ \Delta s^{\text{cor}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\Delta X^{\text{aff}} \Delta S^{\text{aff}} e + \sigma \mu e \end{bmatrix} \tag{2.20}$$

The predictor and corrector directions are then added together to form the predictor-corrector direction

$$(\Delta x, \Delta y, \Delta s) = (\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta s^{\text{aff}}) + (\Delta x^{\text{cor}}, \Delta y^{\text{cor}}, \Delta s^{\text{cor}}) \tag{2.21}$$

The centering parameter, σ, is chosen by using a heuristic [Meh92], which seems to work well in practice. It is defined as

$$\sigma = \left(\frac{\mu^{\text{aff}}}{\mu} \right)^3 \tag{2.22}$$

where

$$\mu^{\text{aff}} = \frac{(x + \alpha_p^{\text{aff}} \Delta x^{\text{aff}})^T (s + \alpha_d^{\text{aff}} \Delta s^{\text{aff}})^T}{n} \tag{2.23}$$

and α_p^{aff} and α_d^{aff} are the primal and dual step length computed for the affine-scaling direction, respectively.

The full algorithm is summarized in Listing 2.2 on the following page.

Listing 2.2: Mehrotra's predictor-corrector algorithm for linear optimization problems in the standard form

Given (x_0, y_0, s_0) and η
for $k = 0, 1, 2, \ldots$ and not converged
  Compute affine direction $(\Delta x_k^{\text{aff}}, \Delta y_k^{\text{aff}}, \Delta s_k^{\text{aff}})$ by solving (2.16)
  Compute step length α_p^{aff} and α_d^{aff} such that
    $x_k^{\text{aff}} + \alpha_p^{\text{aff}} \Delta x_k^{\text{aff}} > 0$ and $s_k^{\text{aff}} + \alpha_d^{\text{aff}} \Delta s_k^{\text{aff}} > 0$.
  Compute centering parameter σ according to (2.22)
  Compute corrector direction $(\Delta x_k^{\text{cor}}, \Delta y_k^{\text{cor}}, \Delta s_k^{\text{cor}})$ by solving (2.20)
  Compute predictor−corrector $(\Delta x_k, \Delta y_k, \Delta s_k)$ according to 2.21
  Compute step length α_k such that
    $x_k + \alpha_k \Delta x_k > 0$ and $s_k + \alpha_k \Delta s_k > 0$.
  Take step
    $(x_{k+1}, y_{k+1}, s_{k+1}) = (x_k, y_k, s_k) + \eta \alpha_k (\Delta x_k, \Delta y_k, \Delta s_k)$
end

2.3.4 Practical considerations

Below we will describe some of the practical considerations when implementing the primal-dual interior point method.

2.3.4.1 Step length

As mentioned, the computed Newton directions does not account for the inequality constraint $(x, s) > 0$. In the long-step algorithm, it is rarely possible to take a full step without violating this inequality constraint [NW06]. Instead, a step length is computed to ensure x and s remain strictly positive and a damped Newton step is taken as shown in (2.9), (2.25) and (2.26).

The primal and dual step length is computed such that

$$x + \alpha_p \Delta x > 0 \quad \text{and} \quad s + \alpha_d \Delta s > 0 \tag{2.24}$$

where α_p is the primal step length and α_d is the dual step length.

The interior point method can then take the step

$$(x_{k+1}) = (x_k) + \eta \alpha_p (\Delta x_k)(y_{k+1}, s_{k+1}) = (y_k, s_k) + \eta \alpha_d (\Delta y_k, \Delta s_k) \tag{2.25}$$

to compute the next iterate, where η is the scaling parameter ensuring the iterate stays within the wide neighborhood around the central path.

Listing 2.3: Initial point computation

$\bar{x} = A^T(AA^T)^{-1}b\,,\ \ \bar{y} = (AA^T)^{-1}(Ag)\,,\ \ \bar{s} = g - A^T\bar{y}$
$\delta_x = \max(-\frac{3}{2}\min(\bar{x}), 0)\,,\ \ \delta_s = \max(-\frac{3}{2}\min(\bar{s}), 0)$
$\hat{x} = \bar{x} + \delta_x e\,,\ \ \hat{s} = \bar{s} + \delta_s e$
$\delta_{\hat{x}} = \frac{1}{2}\frac{\hat{x}^T\hat{s}}{e^T\hat{s}}\,,\ \ \delta_{\hat{s}} = \frac{1}{2}\frac{\hat{x}^T\hat{s}}{e^T\hat{x}}$
$x_0 = \hat{x} + \delta\hat{x}e\,,\ \ y_0 = \bar{y}\,,\ \ s_0 = \hat{s} + \delta\hat{s}e$

An alternative strategy is to take a step with the same step length in the primal and dual direction such as

$$(x_{k+1}, y_{k+1}, s_{k+1}) = (x_k, y_k, s_k) + \eta\alpha_k(\Delta x_k, \Delta y_k, \Delta s_k) \tag{2.26}$$

where $\alpha_k = \min(\alpha_p, \alpha_d)$. This is known as applying the fraction to the boundary rule, which leads to a guaranteed reduction in primal and dual infeasibility [CN07].

2.3.4.2 Initial point

While it is possible to choose any initial point (x_0, y_0, s_0), where $(x_0, s_0) > 0$, for the infeasible method, this is generally a bad idea. A good initial point is critical for the practical performance of the interior point method. However, choosing a good initial point in the primal-dual interior point method is difficult. A commonly used method is the Mehrotra's initial point heuristic [Meh92]. It solves the two least squares problems

$$\begin{aligned} &\min_x x^T x \quad \text{s.t.} \quad Ax = b \\ &\min_{y,s} s^T s \quad \text{s.t.} \quad A^T y + s = c \end{aligned} \tag{2.27}$$

which attempts to satisfy the primal and dual equality constraints. The complementarity pair of the computed solution is then shifted away from the boundary. Listing 2.3 summarizes the algorithm.

2.3.4.3 Termination criteria

The termination criteria for the interior point method is defined as

$$\frac{\|r_p\|}{1 + \|b\|} \leq \text{tol}_p \quad \text{and} \quad \frac{\|r_d\|}{1 + \|c\|} \leq \text{tol}_d \quad \text{and} \quad \frac{x^T s/n}{1 + |c^T x|} \leq \text{tol}_o \tag{2.28}$$

which is also used in [Gon12b].

The first term measures the relative primal infeasibility, the second term measures dual infeasibility, and the last term measures optimality.

2.3.4.4 Computing the Newton direction

The main computational task in interior point methods are solving the linear system of equations in (2.16) for some right-hand side to compute a search direction. The linear system of equations in (2.16) with a general right-hand side is written as

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \tag{2.29}$$

This system can be reduced by eliminating the third equation by computing $\Delta s = X^{-1}(h - S\Delta x)$, resulting in the augmented system form

$$\begin{bmatrix} -\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} f' \\ g' \end{bmatrix} = \begin{bmatrix} f - X^{-1}h \\ g \end{bmatrix} \tag{2.30}$$

where $\Theta = XS^{-1}$. The augmented system is a symmetric indefinite system, which can be solved with factorizations such as Bunch-Parlett factorization [AGMX96] or iterative methods such as MINRES or LSQR [CNW09].

As Θ is a diagonal matrix, its inverse is also diagonal and the augmented systems form can be further reduced by eliminating the first equation by computing $\Delta x = D(A^T\Delta y - f')$, resulting in the normal equations form

$$AD^{-1}A^T\Delta y = g' + AD^{-1}(X^{-1}h - f') \tag{2.31}$$

where $D = \Theta^{-1}$. The matrix $AD^{-1}A^T$ is a positive-definite symmetric matrix, which we will refer to as the normal equations matrix for the standard form. It can be factorized using the Cholesky factorization to solve the linear system. Alternatively, iterative methods for positive-definite symmetric systems can be used such as conjugate gradient [She94].

Chapter 3

Economic Power Plant Portfolio Test Case

In this chapter, we introduce the linear optimization problem from model predictive control (MPC), which we use as our test case throughout the thesis to evaluate the performance of our solvers. The test case is a simplified problem of the power portfolio control problem in [ESJ09]. It is a linear economic MPC problem, where a number of power plants must be controlled to minimize the cost of producing enough power to satisfy the power demand.

We describe the control objective, the system model and the constraints, and demonstrate how a model predictive control problem can be formulated as an optimization problem in inequality and standard form. Finally, we show a solution example and define the model parameters used for the system.

3.1 Description

Our test case describes the power production from N_p power plants. The power production of each of the individual power plants is given by

$$Y_i(s) = G_i(s)U_i(s) \tag{3.1}$$

where $i = 1, \ldots, N_p$, $U_i(s)$ is the input, and $Y_i(s)$ is the output for the power plant i. The transfer function of a power plant is

$$G_i(s) = \frac{1}{(\tau_i s + 1)^3} \tag{3.2}$$

where τ_i is the time constant for power plant i. The transfer function model used for the power plants is described in further detail in [HEBJ10]. The total power production for all the power plants can then be written as a multiple-input single-output (MISO) system as

$$Y(s) = \sum_{i=1}^{N_p} Y_i(s) \tag{3.3}$$

The linear time-invariant discrete state model of the MISO problem is written as

$$x_{k+1} = Ax_k + Bu_k \tag{3.4a}$$
$$y_k = Cx_k \tag{3.4b}$$

where the vectors x_k, y_k, and u_k are the state, power production and input of all the power plants at the discrete time step k, respectively. Given a power demand, r, over a finite horizon of N_t time steps, the power plants must produce enough power to satisfy the demand at any time step k, such that

$$y_k \geq r_k \tag{3.5}$$

for $k = 0, \ldots, N_t$. Each of the power plants also has a minimum and maximum bound on its input, which defines the operating range of the power plant, and a rate-of-movement constraint, which limits the change in the input signal per time step. The input bound constraint is written as

$$u_{\min,k} \leq u_k \leq u_{\max,k} \tag{3.6}$$

where $u_{\min,k}$ and $u_{\max,k}$ are the lower and upper bound on each of the power plants inputs at time step k, respectively. The rate-of-movement constraint on the input is written as

$$\Delta u_{\min,k} \leq \Delta u_k \leq \Delta u_{\max,k} \tag{3.7}$$

where $\Delta u_k = u_k - u_{k-1}$, and $\Delta u_{\min,k}$ and $\Delta u_{\max,k}$ are the maximum negative and positive change in the control input at time step k, respectively.

The full optimization problem with a prediction horizon of N_t discrete time

steps is then written as

$$\begin{aligned}
\min_{u} \quad & \phi = \sum_{k=0}^{N_t-1} c_{u,k}^T u_k && && \text{(3.8a)}\\
\text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k && k = 0, \ldots, N_t - 1 && \text{(3.8b)}\\
& y_k = Cx_k && k = 1, \ldots, N_t && \text{(3.8c)}\\
& u_{\min,k} \leq u_k \leq u_{\max,k} && k = 0, \ldots, N_t - 1 && \text{(3.8d)}\\
& \Delta u_{\min,k} \leq \Delta u_k \leq \Delta u_{\max,k} && k = 0, \ldots, N_t - 1 && \text{(3.8e)}\\
& y_k \geq r_k && k = 1, \ldots, N_t && \text{(3.8f)}
\end{aligned}$$

where $c_{u,k}$ is the cost of power for each power plant at time step k.

Depending on the power demand, r_k, it may not always be possible to satisfy it. In order to ensure the problem is always feasible, we introduce the slack variables s_k which represent buying power from an external provider when the power plants cannot supply enough power to meet the demand. The optimization problem is then written as

$$\begin{aligned}
\min_{u,s} \quad & \phi = \sum_{k=0}^{N_t-1} c_{u,k}^T u_k + \sum_{k=0}^{N_t} c_{s,k}^T s_k && && \text{(3.9a)}\\
\text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k && k = 0, \ldots, N_t - 1 && \text{(3.9b)}\\
& y_k = Cx_k && k = 1, \ldots, N_t && \text{(3.9c)}\\
& u_{\min,k} \leq u_k \leq u_{\max,k} && k = 0, \ldots, N_t - 1 && \text{(3.9d)}\\
& \Delta u_{\min,k} \leq \Delta u_k \leq \Delta u_{\max,k} && k = 0, \ldots, N_t - 1 && \text{(3.9e)}\\
& y_k + s_k \geq r_k && k = 1, \ldots, N_t && \text{(3.9f)}\\
& s_k \geq 0 && k = 1, \ldots, N_t && \text{(3.9g)}
\end{aligned}$$

where $c_{s,k}$ is the cost of buying power at time step k. The cost, $c_{s,k}$, is set to a large value, such that $c_{s,k} \gg \max(c_{u,k})$, to avoid buying external power whenever possible. This is the optimization problem which we will use to test our solvers.

3.2 Formulating in the inequality form

The test problem (3.9) can be written in inequality form as

$$\begin{aligned} \min_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax \geq b \end{aligned} \tag{3.10}$$

where

$$c = \begin{bmatrix} c_u \\ c_s \end{bmatrix} \quad x = \begin{bmatrix} u \\ s \end{bmatrix} \quad A = \begin{bmatrix} I & 0 \\ -I & 0 \\ 0 & I \\ \Psi & 0 \\ -\Psi & 0 \\ \Gamma & I \end{bmatrix} \quad b = \begin{bmatrix} u_{\min} \\ -u_{\max} \\ 0 \\ b_l \\ -b_u \\ r - \Phi x_0 \end{bmatrix} \tag{3.11}$$

and

$$\begin{gathered} u = \begin{bmatrix} u_0 \\ \vdots \\ u_{N_t-1} \end{bmatrix} \quad u_{\min} = \begin{bmatrix} u_{\min,0} \\ \vdots \\ u_{\min,N_t-1} \end{bmatrix} \quad u_{\max} = \begin{bmatrix} u_{\max,0} \\ \vdots \\ u_{\max,N_t-1} \end{bmatrix} \\ \Delta u = \begin{bmatrix} \Delta u_0 \\ \vdots \\ \Delta u_{N_t-1} \end{bmatrix} \quad \Delta u_{\min} = \begin{bmatrix} \Delta u_{\min,0} \\ \vdots \\ \Delta u_{\min,N_t-1} \end{bmatrix} \quad \Delta u_{\max} = \begin{bmatrix} \Delta u_{\max,0} \\ \vdots \\ \Delta u_{\max,N_t-1} \end{bmatrix} \\ s = \begin{bmatrix} s_0 \\ \vdots \\ s_{N_t} \end{bmatrix} \quad y = \begin{bmatrix} y_0 \\ \vdots \\ y_{N_t} \end{bmatrix} \quad r = \begin{bmatrix} r_0 \\ \vdots \\ r_{N_t} \end{bmatrix} c_u = \begin{bmatrix} c_{u,0} \\ \vdots \\ c_{u,N_t-1} \end{bmatrix} \quad c_s = \begin{bmatrix} c_{u,0} \\ \vdots \\ c_{u,N_t} \end{bmatrix} \end{gathered} \tag{3.12}$$

In the following sections, we describe the structure of the sub-matrices and how the constraints are converted to matrix form.

3.2.1 Bound constraints

The bound constraints on the input and the slack variables in (3.9) are

$$u_{\min} \leq u_k \leq u_{\max} \qquad k = 0, \ldots, N_t - 1 \tag{3.13}$$

$$s_k \geq 0 \qquad k = 1, \ldots, N_t \tag{3.14}$$

In inequality form, each of these can simply be written with an identity matrix as

$$\begin{bmatrix} I & 0 \\ -I & 0 \\ 0 & I \end{bmatrix} \overbrace{\begin{bmatrix} u \\ s \end{bmatrix}}^{x} \geq \begin{bmatrix} u_{\min} \\ -u_{\max} \\ 0 \end{bmatrix} \tag{3.15}$$

3.2.2 Rate of movement constraints

The rate-of-movement constraints on the input in (3.9) are

$$\Delta u_{\min} \leq u_k - u_{k-1} \leq \Delta u_{\max} \qquad k = 0, \ldots, N_t - 1 \tag{3.16}$$

To write these as a matrix, we introduce the matrix Ψ which is written as

$$\Psi = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ -I & I & 0 & 0 & 0 \\ 0 & \ddots & \ddots & 0 & 0 \\ 0 & 0 & \ddots & \ddots & 0 \\ 0 & 0 & 0 & -I & I \end{bmatrix} \tag{3.17}$$

where I is the identity matrix with the dimensions $N_p \times N_p$. The rate-of-movement constraint are then written as

$$\overbrace{\begin{bmatrix} \Delta u_{\min,0} + u_{-1} \\ \Delta u_{\min,1} \\ \vdots \\ \Delta u_{\min,N_t-1} \\ \Delta u_{\min,N_t} \end{bmatrix}}^{b_l} \leq \begin{bmatrix} \Psi & 0 \end{bmatrix} \overbrace{\begin{bmatrix} u \\ s \end{bmatrix}}^{x} \geq \overbrace{\begin{bmatrix} \Delta u_{\max} + u_{-1} \\ \Delta u_{\max,1} \\ \vdots \\ \Delta u_{\max,N_t-1} \\ \Delta u_{\max,N_t} \end{bmatrix}}^{b_u}$$

which also can be written as

$$\begin{bmatrix} \Psi & 0 \\ -\Psi & 0 \end{bmatrix} x \geq \begin{bmatrix} b_l \\ -b_u \end{bmatrix}$$

3.2.3 Power demand constraint

The dynamics of the power plant model can be written as a finite impulse response (FIR) model, which is commonly used in MPC, as it can represent any

kind of stable dynamic process [Edl10].

Given the discrete state space

$$x_{k+1} = Ax_k + Bu_k \qquad k = 0, \ldots, N_t - 1 \tag{3.18}$$
$$y_k = Cx_k \qquad k = 1, \ldots, N_t \tag{3.19}$$

the FIR model is written as

$$y_k = CA^k x_0 + \sum_{i=0}^{k-1} H_i u_i \quad k = 1, \ldots, N_t \tag{3.20}$$

where $H_i = CA^{i-1}B$. This follows from straight-forward substitution of (3.18) in (3.19). The FIR model can now be written in matrix form as

$$y = \Phi x_0 + \Gamma u \tag{3.21}$$

where

$$\Phi = \begin{bmatrix} CA \\ \vdots \\ CA^{N_t} \end{bmatrix} \qquad \Gamma = \begin{bmatrix} H_1 & 0 & 0 & 0 \\ H_2 & H_1 & 0 & 0 \\ \vdots & \ddots & \ddots & 0 \\ H_{N_t} & \ldots & H_2 & H_1 \end{bmatrix} \tag{3.22}$$

This eliminates all the internal states, except the initial one x_0, when computing the output y_k. The FIR model (3.21) is then substituted into our power demand constraint

$$y_k + s_k \geq r_k \qquad k = 1, \ldots, N_t \tag{3.23}$$

such that it is written as

$$\begin{bmatrix} \Gamma & I \end{bmatrix} \overbrace{\begin{bmatrix} u \\ s \end{bmatrix}}^{x} \geq \begin{bmatrix} r - \Phi x_0 \end{bmatrix}$$

3.2.4 Solution example

Figure 3.1 shows an example of the power plant portfolio problem with a prediction horizon of 500 time steps and 2 power plants. The top graph shows the power demand, r, also known as the reference, and the total power production. The other two graphs shows the control signal in red and the power production

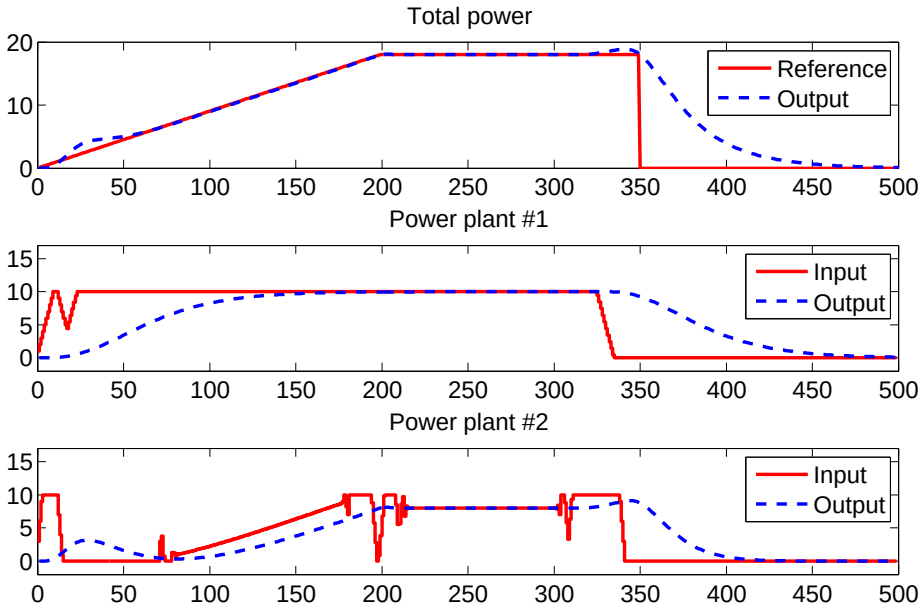


Figure 3.1: Solution example of the power plant portfolio problem with two power plants. Power plant 1 is slow, but cheap, while power plant 2 is fast, but expensive.

	P.G. #1	P.G. #2
τ	20	10
$u_{\min}$	0	0
$u_{\max}$	10	10
$\Delta u_{\min}$	-1	-3
$\Delta u_{\max}$	1	3
c_u	1	2

Table 3.1: Parameters used for solution example with a cheap and slow power plant and an expensive and fast power plant. The cost of the slack variables, c_s, is set to 10^5. While the test case allow for time-variant bounds and cost, we use the same value for all time steps.

output in blue for each of the two power plants. Table 3.1 shows the parameters used to model the two power plants.

The first power plant is set to have a smaller cost than the second power plant, while its time constant is higher. This results in a cheap, but slow power plant and an expensive, but fast power plant. These parameters were also used in [HEBJ10].

The solution of the problem shows that the controller tries to use the cheap, but slow, power plant as much as possible to satisfy the power demand. The fast, but expensive, power plant is used in the beginning to satisfy the power demand quickly as well as during changes in the reference to quickly change the power production to avoid buying power from an external provider, which is much more expensive.

3.3 Formulating in the standard form

The standard form of a linear optimization problem is written as

$$\begin{aligned} \min_{x} \quad & \phi_p = c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{3.24}$$

The inequality form of the test case problem shown in (3.10) can also be written in the standard form by introducing slack variables, s, such that the inequality constraints

$$Ax \geq b \tag{3.25}$$

are written as

$$\begin{aligned} Ax - s &= b \\ s &\geq 0 \end{aligned} \tag{3.26}$$

where $x \geq 0$ and $s \geq 0$. We can write this in matrix form as

$$\hat{A}\hat{x} = b \tag{3.27}$$

where $\hat{x} \geq 0$, $\hat{A} = \begin{bmatrix} A & -I \end{bmatrix}$ and $\hat{x} = \begin{bmatrix} x \\ s \end{bmatrix}$.

The standard form of our test problem is then written as

$$\begin{aligned} \min_{\hat{x}} \quad & \hat{\phi}_p = \hat{c}^T \hat{x} \\ \text{s.t.} \quad & \hat{A}\hat{x} = b \\ & \hat{x} \geq 0 \end{aligned} \tag{3.28}$$

where $\hat{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}$.

3.4 Summary

We introduced a test case based on an economic power plant portfolio system from model predictive control (MPC). The test case is used throughout the thesis to evaluate the performance of our solvers and serves as a good general MPC problem as it has input bound constraints, rate-of-movement constraints, and output bound constraints by using a discrete state space model formulated as a finite impulse response model.

The test case problem was formulated as an optimization problem in both inequality form and standard form, which are common general representations of optimization problems. We will base our solvers on these two common forms.

Chapter 4

Interior Point Method for Linear Optimization Problems in Inequality Form on GPU

In this chapter, we describe the design and implementation of a primal-dual interior point method for linear optimization problems in the inequality form. The algorithm is implemented with a direct solver to compute Newton directions by forming the normal equations and factorizing the symmetric positive-definite normal equations matrix with Cholesky factorization.

We have implemented both a generic version of the solver, and a problem-specific version of the solver which exploits the unique problem structure for our dynamical optimization test case from Chapter 3.

Both versions of the solver have been implemented in MATLAB, MATLAB with built-in GPU support, and in C with CUDA. Their performance is measured and compared to determine whether utilizing GPUs can reduce the solution time of the interior point method.

4.1 Method

The method described in this chapter is based on the interior point method for model predictive control described in [ESJ09]. It is a standard primal-dual interior point method with Mehrotra predictor-corrector for solving a linear optimization problem in the inequality form

$$\begin{aligned} \min_{x} \quad & f(x) = c^T x \\ \text{s.t.} \quad & Ax \geq b \end{aligned} \tag{4.1}$$

Designing the interior point method to specifically solve an optimization problem in the inequality form results in slightly different KKT conditions than solving an optimization problem in the standard form.

The Lagrangian of the optimization problem in the inequality form is given by

$$L(x, y) = c^T x - y^T (Ax - b) \tag{4.2}$$

which results in the following first order necessary conditions

$$\nabla_x L(x, y) = c - A^T y = 0 \tag{4.3}$$
$$\nabla_y L(x, y) = Ax - b \geq 0 \perp y \geq 0 \tag{4.4}$$

where $\perp$ is used to denote complementarity. By introducing the slack variable $s = Ax - b$, we can write the conditions as

$$c - A^T y = 0 \tag{4.5}$$
$$Ax - s = b \tag{4.6}$$
$$s \geq 0 \perp y \geq 0 \tag{4.7}$$

which can then be written in matrix form as

$$\begin{bmatrix} 0 & A^T & 0 \\ A & 0 & -I \\ 0 & S & Y \end{bmatrix} \begin{bmatrix} x \\ y \\ s \end{bmatrix} = \begin{bmatrix} c \\ b \\ 0 \end{bmatrix} \tag{4.8}$$

where $S = diag(s)$ and $Y = diag(y)$. Note that (4.8) is slightly different than the KKT conditions for the standard form shown in (2.7). The slack variable s is applied to the primal in the inequality formulation, resulting in s and y being the complementarity pair, while in the standard form the slack variable s is applied to the dual with x and s being the complementarity pair.

4.1.1 Computing the Newton direction

The Newton direction with a general right-hand side in the interior point method for linear optimization problems in the inequality form is written as

$$\begin{bmatrix} 0 & A^T & 0 \\ A & 0 & -I \\ 0 & S & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \tag{4.9}$$

As in the standard formulation, described in Section 2.3.4.4, we can eliminate the third equation by computing $\Delta s = Y^{-1}(h - S\Delta y)$ to reduce it to the augmented systems form

$$\begin{bmatrix} 0 & A^T \\ A & \Theta \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} f' \\ g' \end{bmatrix} = \begin{bmatrix} f + Y^{-1}h \\ g \end{bmatrix} \tag{4.10}$$

where $\Theta = S^{-1}Y$. This can be further reduced by eliminating the second equation by computing $\Delta y = D(g' - A\Delta x)$, resulting in the normal equations form

$$A^T DA\Delta x = A^T Dg' - f' \tag{4.11}$$

where $D = \Theta^{-1}$. In this chapter, we will refer to $A^T DA$ as the normal equations matrix.

In the implementations in this chapter, we compute the Newton directions by solving the normal equations form with a direct method, particularly Cholesky factorization. There are multiple benefits to solving the normal equations system with a direct method. Solving 4.11 with direct methods is stable and the effect of the ill-conditioning caused by the spread in the complementarity matrix Θ is minimal [Gon12b].

Additionally, the factorization may be used to compute both the affine step and the corrector step in Mehrotra's predictor-corrector algorithm [Meh92], which we will describe in Section 4.1.4 for the inequality form. This means the cost of computing both Newton directions is almost the same as computing one of them. For further details of the advantages and disadvantages of solving the normal equations form, we refer to [AGMX96].

4.1.2 Initial point

Mehrotra's initial point heuristic [Meh92], which we presented in Section 2.3.4.2 for the interior point method for linear optimization problems in the standard form, is also used in our implementation of the interior point method for linear

Listing 4.1: Initial point computation

$\bar{y} = A(A^TA)^{-1}c\,, \quad \bar{x} = (A^TA)^{-1}(A^Tb)\,, \quad \bar{s} = b - A\bar{x}$
$\delta_y = \max(-\frac{3}{2}\min(\bar{y}), 0)\,, \quad \delta_s = \max(-\frac{3}{2}\min(\bar{s}), 0)$
$\hat{y} = \bar{y} + \delta_y e\,, \quad \hat{s} = \bar{s} + \delta_s e$
$\delta_{\hat{y}} = \frac{1}{2}\frac{\hat{y}^T\hat{s}}{e^T\hat{s}}\,, \quad \delta_{\hat{s}} = \frac{1}{2}\frac{\hat{y}^T\hat{s}}{e^T\hat{y}}$
$x_0 = \bar{x}\,, \quad y_0 = \hat{y} + \delta\hat{y}e\,, \quad s_0 = \hat{s} + \delta\hat{s}e$

optimization problems in the inequality form. However, we have modified it corresponding to the change in the KKT conditions. The computation of the initial point is shown in Listing 4.1.

4.1.3 Termination criteria

The termination criteria in (2.28) is adjusted to correspond to the slightly different KKT system, where s and y are the complementarity pair instead. This results in the following termination criteria

$$\frac{\|r_p\|}{1+\|b\|} \leq tol_p \quad \text{and} \quad \frac{\|r_d\|}{1+\|c\|} \leq tol_d \quad \text{and} \quad \frac{s^Ty/m}{1+|c^Tx|} \leq tol_o \tag{4.12}$$

4.1.4 Algorithm

The algorithm used in [ESJ09], which we implement in this chapter, is based on Mehrotra's predictor-corrector algorithm [Meh92]. We presented this algorithm for the interior point method for linear optimization problems in standard form in Section 2.3.3.

The affine Newton direction for the interior point method for linear optimization problems in the inequality form is computed by solving

$$\begin{bmatrix} 0 & A^T & 0 \\ A & 0 & -I \\ 0 & S & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} r_d \\ r_p \\ r_c \end{bmatrix} = \begin{bmatrix} c - A^Ty \\ b - Ax + s \\ -SYe \end{bmatrix} \tag{4.13}$$

The full algorithm used for our implementations in this chapter is shown in Listing 4.2. The Newton directions in the algorithm are solved for a general right-hand side according to (4.9) by solving the normal equations form (4.11) using Cholesky factorization to factorize the normal equations matrix, $A^TDA =$

Name	Implemented in	GPU usage
LPippdIneq	MATLAB	No
LPippdIneqGPUv1	MATLAB	Cholesky
LPippdIneqGPUv2	MATLAB	Matrix multiplication + Cholesky
LPippdIneqGPUv3	MATLAB	Matrix multiplication + Cholesky
LPippdIneqGPUEx	C/CUDA	All

Table 4.1: Generic solver implementations

LL^T. We use $\eta = 0.99995$ for our tests and m is the number of elements in the complementarity pair.

4.2 Generic implementation

The generic implementation of the primal-dual interior point method does not assume any structure for the A matrix, and it is applicable for any optimization problem stated as 4.1. We have implemented the algorithm in plain MATLAB, in MATLAB with GPU-acceleration and in C with GPU acceleration. In this section, we describe the various implementations and the obtained results. An overview of the implementations is shown on Table 4.1.

4.2.1 Plain MATLAB implementation

The implementation of the algorithm in plain MATLAB works as a reference for the other versions in this thesis, and it is a straight-forward translation of the algorithm in Listing 4.2 to MATLAB. There are two things to note about this implementation, which are also important for the remaining MATLAB implementations. The interface for the function is shown in Listing 4.3.

Listing 4.3: MATLAB solver interface

```
function [x, status, y, info, history] = LPippdIneq(g,A,b,options)
```

Firstly, the A matrix can be either a matrix or a structure. If A is a structure, then the algorithm expects the fields `A.funAx` and `A.funAtx` to exist and be function pointers to matrix-vector multiplication and matrix-transpose-vector multiplication with A, respectively. This allows the user to implement efficient

Listing 4.2: Mehrotra's predictor-corrector algorithm for linear optimization problems in the inequality form with direct solver

Input: $\eta\,,\ k_{max}$
Initial point: Initialize (x,y,s) according to Listing 4.1
Residuals: $r_d = c - A^T y\,,\ r_p = b - Ax + z\,,\ r_c = -SYe$
Duality measure: $\mu = \frac{s^T y}{m}$
Iteration counter: $k = 0$
while $(k < k_{max}$ and $\frac{\|r_p\|}{1+\|b\|} \leq tol_p$ and $\frac{\|r_d\|}{1+\|c\|} \leq tol_d$ and $\frac{s^T y/m}{1+|c^T x|} \leq tol_o)$
  $D = S^{-1}Y$
  Cholesky factorization: $A^T DA = LL^T$
  Affine step:
    $f^{\text{aff}} = r_d\,,\ g^{\text{aff}} = r_p\,,\ h^{\text{aff}} = r_c$
    $r^{\text{aff}} = A^T D g^{\text{aff}} - (f^{\text{aff}} + Y^{-1}h^{\text{aff}})$
    $\Delta x^{\text{aff}} = L^T \backslash (L \backslash (r^{\text{aff}}))\,,$
    $\Delta y^{\text{aff}} = D(f^{\text{aff}} - A\Delta x)$
    $\Delta s^{\text{aff}} = Y^{-1}(h^{\text{aff}} - S\Delta y)$
    Compute affine step length α_p^{aff} and α_d^{aff} such that $s + \alpha_p^{\text{aff}} \Delta s^{\text{aff}} \geq 0$ and $s + \alpha_d^{\text{aff}} \Delta s \geq 0\,.$
  Affine duality measure: $\mu^{\text{aff}} = \frac{(s + \alpha_p^{\text{aff}} \Delta s^{\text{aff}})^T (y + \alpha_d^{\text{aff}} \Delta y^{\text{aff}})^T}{m}$
  Corrector step:
    $\sigma = \left(\frac{\mu^{\text{aff}}}{\mu}\right)^3$
    $f^{\text{cor}} = 0\,,\ g^{\text{cor}} = 0\,,\ h^{\text{cor}} = -\Delta S^{\text{aff}} \Delta Y^{\text{aff}} e + \sigma\mu e$
    $r^{\text{cor}} = A^T D g^{\text{cor}} - (f^{\text{cor}} + Y^{-1}h^{\text{cor}})$
    $\Delta x^{\text{cor}} = L^T \backslash (L \backslash (r^{\text{cor}}))\,,$
    $\Delta y^{\text{cor}} = D(f^{\text{cor}} - A\Delta x)$
    $\Delta s^{\text{cor}} = Y^{-1}(h^{\text{cor}} - S\Delta y)$
  Predictor-corrector step:
    $(\Delta x, \Delta y, \Delta s) = (\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta s^{\text{aff}}) + (\Delta x^{\text{cor}}, \Delta y^{\text{cor}}, \Delta s^{\text{cor}})$
    Compute step length α such that $s + \alpha \Delta s_k > 0$ and $y + \alpha \Delta y > 0\,.$
  Take step: $(x,y,s) = (x,y,s) + \eta\alpha(\Delta x, \Delta y, \Delta s)$
  Residuals: $r_d = c - A^T y\,,\ r_p = b - Ax + z\,,\ r_c = -SYe$
  Duality measure: $\mu = \frac{s^T y}{m}$
  Iteration counter: $k = k + 1$
end

Listing 4.4: Default Cholesky-based normal equations solver

```
function [x, data] = SolveNE(k, A, D, b, data)
  if (isfield(data,'k') && k == data.k && isfield(data,'L'))
    L = data.L;
  else
    H = A.funH(A,D);
    L = chol(H,'lower');
    data.k = k; data.L = L;
  end
  x = L'\(L\(b));
end
```

problem-specific functions for the two operations. If A is passed as a matrix, then the algorithm will automatically convert it to a structure by storing the A matrix in it and defining the necessary functions.

Secondly, the `options` may contain a field called `solverNE`. This field is per default set to `'chol'`, which means the algorithm will use its built-in Cholesky-based normal equations solver which is shown in Listing 4.4. The default normal equations solver computes the normal equations matrix, factorizes the matrix and then solves for $Hx = b$. It uses the *data* parameter for data persistence and reuses the factorization of the normal equations matrix if it has already been computed within the same iteration. Additionally, if the default normal equations solver is used and A is passed as a structure to the algorithm, then the field `A.funH` must be a function pointer to a function which returns the normal equations matrix.

This parameter allows the user to overwrite the default normal equations solver by setting the `solverNE` field in `options` to a function handle with the same interface as `SolveNE` above. We will use this later to implement the GPU-accelerated and problem-specific solvers in MATLAB without modifying the original implementation.

4.2.2 MATLAB with GPU

Since MATLAB version R2010a, MATLAB has added support for some GPU operations on NVIDIA graphic cards through CUBLAS. Unfortunately, not all MATLAB operations are supported and, while it is easy to use, it can be difficult to specify operations in MATLAB, which optimally utilize the GPU. Furthermore, the built-in GPU support in MATLAB only supports dense matrices as

of R2013b, which means sparse matrices must be converted to a dense matrix to operate with them on the GPU.

Much more advanced GPU support existed for MATLAB in the form of the proprietary 3rd-party add-on Jacket from AccelerEyes, including sparse matrix operations. However, due to license disagreements it is no longer available for sale since December 2012 [Acc12]. MathWorks and AccelerEyes have stated that it will eventually be incorporated directly into the Parallel Computing Toolbox, but at present it is still unavailable.

For this reason, we have only utilized the built-in GPU support in MATLAB in the implementation of our GPU-accelerated MATLAB solver. The solver is implemented in three different versions to determine the best way to extend the MATLAB implementation with GPU acceleration.

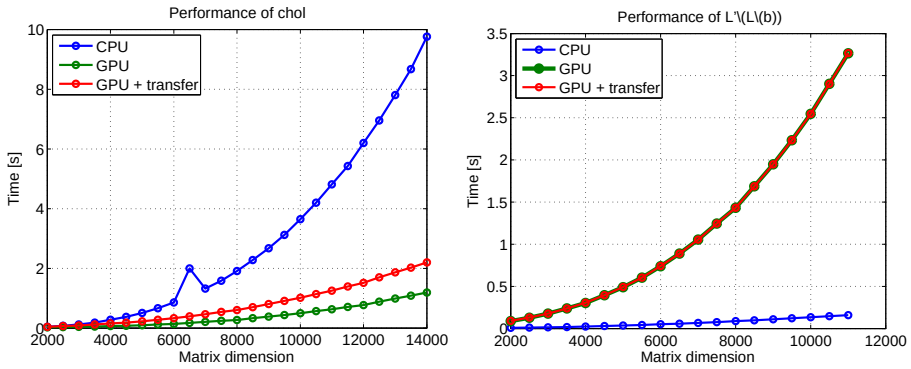
4.2.2.1 Benchmark of MATLAB GPU functions

We have tested the MATLAB implementation of Cholesky factorization and triangular solve on the GPU to determine if we can accelerate our normal equations solver with MATLAB's built-in GPU support. We ran Cholesky factorization on a positive-definite symmetric dense matrix with the dimensions 2000×2000 and incremented the dimensions by 500 until we ran out of memory on the GPU. The computed factor was then used to solve $L^T \backslash (L \backslash b)$. The test was done on the test system mentioned in Section 1.4 on page 4. The results are shown on Figure 4.1a and Figure 4.1b for the two operations.

The Cholesky factorization of a dense matrix can be accelerated substantially by using a GPU, even when including the cost of transferring the dense positive-definite matrix to the GPU, factorizing and transferring it back to the CPU.

The triangular solve with the resulting Cholesky factorization is much slower than solving on the CPU. The cost of transferring the vector b to the GPU and transferring the result back does not significantly impact performance, as the vectors are small.

In the following implementations, we will not use triangular solve on the GPU, but instead transfer the result of the Cholesky factorization back to the CPU and do triangular solve on the CPU.



(a) Performance of chol(H,'lower') **(b)** Performance of L'\(L\b)

Figure 4.1: Performance of MATLAB Cholesky factorization and triangular solve on the GPU.

4.2.2.2 Version 1

In this version, only the Cholesky factorization is computed on the GPU, since it is the most time-consuming operation. First, we compute the normal equations matrix, H, on the CPU, using sparse matrix-matrix multiplication. Then, we transfer the resulting normal equations matrix to the GPU, call Cholesky factorization on the GPU-resident matrix resulting in GPU-based factorization, and then transfer the lower triangular matrix back to the CPU, to be used for triangular solve on the host.

The implementation of this modified normal equations solver is shown in Listing 4.5. The MATLAB function `gpuArray()` transfers the matrix from the CPU to the GPU and `chol()` is used to factorize the GPU-resident matrix. The result from Cholesky factorization is then transferred back to the CPU by using the MATLAB function `gather()`.

4.2.2.3 Version 2

In this version, we transfer the entire constraints matrix A to the GPU prior to solving and use the GPU for all matrix multiplications with A, including the computation of the normal equations matrix and the Cholesky factorization. Since the A matrix is stored densely on the GPU, as well as the normal equations matrix and the Cholesky factorization, this version is much more memory intensive than version 1. It requires a total of $(m \times n + 2n \times n)$ matrix elements

to be stored on the GPU, while version 1 only requires $(2n \times n)$ elements. As our test problem has $m \gg n$, the addition of $m \times n$ is very expensive.

The only difference in this implementation, compared to version 1, is that it uses `gpuArray()` to transfer and store the $A.matrix$ on the GPU before calling the solver function. This causes all operations with the matrix to be done on the GPU automatically, including matrix-vector and matrix-matrix operations. The code for the equation solver is also almost identical to version 1, so we will not show the implementation here. The difference is that it is no longer necessary to call `gpuArray(H)` as H is computed and stored on the GPU already and that a call to `gather()` has been added to the result of matrix-vector multiplications to copy it back to the CPU.

4.2.2.4 Version 3

In version 2, the computation of the normal equations matrix is done exactly as in version 1. However, since MATLAB does not support sparse matrices on the GPU, the diagonal matrix D is transferred and stored on the GPU as a dense matrix. This requires additional memory to store D and computing a full matrix-matrix multiplication with a diagonal matrix seems inefficient. Multiplying a diagonal matrix D with a general matrix A is equivalent to scaling the row i in A with the corresponding diagonal element i in D. In this version, we replace the computation of H with the code in Listing 4.6, which uses a for-loop to scale the rows in A.

Listing 4.5: MATLAB GPU-based normal equations solver version 1

```
function [x, data] = SolveNEgpuv1(k, A, D, b, data)
  if (isfield(data,'k') && k == data.k && isfield(data,'L'))
    L = data.L;
  else
    H = A.matrix' * spdiags(D,[0],length(D),length(D)) *
        ↪ A.matrix;
    L = gather(chol(gpuArray(full(H)),'lower'));
    data.k = k; data.L = L;
  end
  x = L'\(L\(b));
end
```

Listing 4.6: Computation of H in version 3

```
H = A.matrix;
for i=1:size(A.matrix,1)
  H(i,:) = D(i)*H(i,:);
end
H = A.matrix' * H;
```

4.2.3 C/CUDA implementation

The final generic implementation of the interior point method is done in C with CUDA. We have implemented it using available libraries such as CUBLAS and MAGMA. CUBLAS is a dense BLAS implementation by NVIDIA, which implements all the standard BLAS operations on the GPU. MAGMA is a dense linear algebra library, similar to LAPACK, for heterogeneous architectures, including GPU support. It contains a very fast implementation of Cholesky factorization on NVIDIA GPUs [LTND10].

The implementation is done such that all data is kept on the GPU and all computation is done on the GPU as well. This is to avoid data transfers between the CPU and the GPU, which can be costly. However, this requires the A matrix to be stored on the GPU. Since we use dense matrices on the GPU, this requires a lot of memory, similar to version 2 of the MATLAB implementation. We will describe this further in Section 4.2.3.7.

While most of the operations in the solver can be implemented with calls to CUBLAS and MAGMA, there are a few operations, such as the step length computation, which cannot be computed directly using BLAS and LAPACK operations. For these operations, we have implemented CUDA kernels to compute the result. In the following sections, we will describe some of the implementation details.

Listing 4.7: MATLAB

```
rP = s - A*x + b;
rD = c - A'*y;
rC = -s .* y;
```

Listing 4.8: C/CUDA

```
cublasDgemv('N', M, N, -1.0, A, M, x, 1,
    ↪ 0.0, rP, 1);
cublasDaxpy(M, 1.0, b, 1, rP, 1);
cublasDaxpy(M, 1.0, s, 1, rP, 1);
cublasDgemv('T', M, N, -1.0, A, M, y, 1,
    ↪ 0.0, rD, 1);
cublasDaxpy(N, 1.0, c, 1, rD, 1);
elementMultiplyVector(s, y, rC, M);
```

4.2.3.1 Computing residual vectors

As mentioned above, the use of standard BLAS and LAPACK libraries for GPU makes it easy to implement most of the solver. As an example of this, we show the implementation of the residual vector computation in MATLAB in Listing 4.7 alongside the CUDA implementation in Listing 4.8.

The residual vector r_P is computed by first using `dgemv()` to compute $-Ax$, then `daxpy()` is used to compute $-Ax + b$, and finally `daxpy()` is used again to compute $s - Ax + b$. The residual r_D is computed similarly with a call to `dgemv()` to compute $-A^T y$ and then `daxpy()` to compute $g - A^T y$. Finally, the residual r_C requires element-wise vector multiplication, since the diagonal matrices S and Y are stored as vectors. Element-wise vector multiplication is not available in BLAS, so the function `elementMultiplyVector()` is our own implementation of this operation.

4.2.3.2 Element-wise vector operations

As mentioned in the previous section, we require functions to do element-wise vector operations to work with diagonal matrices stored as vectors. While element-wise vector operations are not available in BLAS, they are rather simple operations, which are easy to implement.

The implementation for element-wise vector multiplication is shown in Listing 4.9.

The function `elementMultiplyVector_kernel()` is the CUDA kernel, which computes the element-wise multiplication. Each thread computes the element corresponding to its global thread index by loading the associated elements from each of the two input vectors, multiplying them, and storing the result in the

Listing 4.9: CUDA code to compute element-wise vector multiplication

```
1  __global__ void elementMultiplyVector_kernel(double* x,
       ↪ double* y, double* z, unsigned int N) {
2    unsigned int idx;
3    for (idx = blockIdx.x * blockDim.x + threadIdx.x; idx < N;
         ↪ idx += gridDim.x * blockDim.x)
4      z[idx] = x[idx] * y[idx];
5  }
6  void elementMultiplyVector(double* x, double* y, double* z,
       ↪ unsigned int N) {
7    dim3 blockDim(128);
8    dim3 gridDim(\min((N-1) / blockDim.x + 1,32*1024));
9    elementMultiplyVector_kernel<<<gridDim, blockDim>>>(x, y,
         ↪ z, N);
10 }
```

output vector. In case there are less threads than the number of elements, N, a loop is added, which increments the index with the total number of threads. This allows threads to work on more than one element, if there are not enough threads.

The function `elementMultiplyVector()` is the interface function, with which we call the CUDA kernel. We define the number of threads per block to 128 and compute the required number of thread blocks based on the length of the vector.

We have implemented similar functions to do element-wise vector division, as well as functions to add a constant to a vector and set all the elements in a vector to the same value. The implementation of these is essentially the same as the element-wise multiplication. The only difference is changing line 4 in Listing 4.9 with a different expression, therefore we will not show them here.

4.2.3.3 Computing the smallest element of a vector

For the computation of the initial point, as described in Listing 4.1, it is necessary to find the smallest element in $\bar{y}$ and $\bar{s}$. While BLAS has a minimum magnitude function, `idamin()`, which finds the index of the element that satisfies $\min(abs(x))$, there is no function to find the minimum of a vector, which has negative elements as well.

Listing 4.10: CUDA code to compute minimum of vector

```
__global__ void computeMinimum_kernel(const unsigned int N,
    ↪ double* x, double* result)
{
  extern __shared__ double sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  // Load into shared memory
  double val = DBL_MAX;
  if (i < N) {
    val = x[i];
  }
  sdata[tid] = val;
  __syncthreads();

  // Find min
  for(unsigned int s = blockDim.x/2; s > 0; s>>=1) {
    if(tid < s) {
      sdata[tid] = f\min(sdata[tid], sdata[tid+s]);
    }
    __syncthreads();
  }

  // Write local minimum
  if (tid == 0)
    result[blockIdx.x] = sdata[0];
}
```

Our CUDA kernel to find the smallest element of a vector is based on a standard sum reduction kernel, which is discussed extensively in [Har07]. The operation to compute the sum of a vector is very similar to finding the minimum, as we simply replace the addition operation with a minimum operation. With this minor modification, it becomes an efficient kernel for finding the minimum. The code is shown in Listing 4.10.

The kernel thread uses its global thread index to determine which element to load. It sets `val` to the maximum possible value and then overwrites with the threads element in the x vector. If the thread index exceeds the number of elements, then `val` is left as the maximum value. This prevents excess threads from affecting the result. The value is stored in shared memory and the thread block is synchronized to ensure all threads in the block have loaded their value into shared memory.

Once all the elements have been loaded, the thread starts a loop, which compares two elements in shared memory and stores the resulting minimum in shared memory, reducing the number of elements to half. The next iteration of the loop compares the elements from the previous iteration, reducing the number of elements to half again. This continues until the minimum element in shared memory is found and the result is written to an output vector corresponding to the thread block index.

As there is no synchronization across the thread blocks in CUDA, the result is that each thread block computes its own minimum for the elements it loaded into shared memory. The resulting array of minima can then either be transferred to the CPU and find the minimum there, or passed to the kernel again to further reduce number of minima, until only one value remains.

4.2.3.4 Computing the step length

The computation of the step length is similar to the computation of the minimum. Instead of finding the minimum element in the entire vector, we must instead find the minimum value of the computation $-\frac{x_i}{\Delta x_i}$ and only for elements i, where $\Delta x_i < 0$. This means that we have a restriction on which elements to include as well as a computation. This can be done by simply replacing the lines 8 to 11 in Listing 4.10 with the code in Listing 4.11.

In this code, we set `val` to one, as this is the step length for elements where $\Delta x_i \geq 0$. Then, we load the corresponding element in Δx and if it is negative, we compute the step length for the element and store it in `val`.

Listing 4.11: CUDA code to compute step length

```
double val = 1.0;
if (i < N) {
  double val_dx = dx[i];
  if (val_dx < 0.0)
    val = -(x[i] / val_dx);
}
```

4.2.3.5 Computing the normal equations matrix

There are two steps involved in computing the normal equations matrix, $H = A^T DA$. The diagonal matrix D must first be multiplied with the constraints matrix A, and then the result is multiplied with the transpose matrix A^T. Multiplying a diagonal matrix D with a general matrix A is equivalent to scaling the row i in A with the corresponding diagonal element i in D. This could be done with a call to `dscal()` for every row, however it would result in a kernel call for every row and the memory access would not be coalesced, since the matrix is stored column-wise. Instead, we have implemented a CUDA kernel to compute DA efficiently. The CUDA implementation is shown in Listing 4.12.

The interface function `diagMatrixMultMatrix()` creates a 2D thread grid with 2D thread blocks of size 32×32, resulting in thread blocks of 512 threads. The number of thread blocks in each dimension of the thread grid is computed by dividing the dimensions of the result matrix with the corresponding thread block dimension, such that there is a thread for each element in the result matrix, if possible. We limit the maximum number of thread blocks in any dimension to 32768 and handle elements beyond that by reusing threads with a loop in the kernel.

For 2D thread blocks, threads are grouped in a warp along the x-dimension. In the kernel, we use the x-dimension to select the row to load, as data is stored column-wise. This causes each thread in a warp to load adjacent values, which results in coalesced reads and writes for optimal performance.

Once DA has been computed with the kernel, the result is multiplied with A^T with a call to `dgemm()`. It is interesting to note that a new function, `cublasDdgmm()`, has been introduced in CUBLAS 5.5, which computes the result of a diagonal matrix multiplied with a general matrix. We have not used this function, as it was not available at the time of our implementation and we would still need to implement the other element-wise operations.

4.2.3.6 Factorization and triangular solve

The Cholesky factorization of a dense matrix and the triangular solve with the resulting factor are both very easy to compute on the GPU by using the MAGMA library. Among its many functions, it contains the LAPACK routines `dpotrf()` and `dpotrs()`, which can be used to factorize and solve respectively.

Listing 4.12: Kernel for multiplying a diagonal matrix with a dense matrix in CUDA

```
__global__ void diagMatrixMultMatrix_kernel(const double* x,
    ↪ const double* A, unsigned int lda, double* B, unsigned
    ↪ int ldb, unsigned int M, unsigned int N) {
  unsigned int i, j;
  for (i = blockIdx.x * blockDim.x + threadIdx.x; i < M; i +=
      ↪ gridDim.x * blockDim.x) {
    double xval = x[i];
    for (j = blockIdx.y * blockDim.y + threadIdx.y; j < N; j
        ↪ += gridDim.y * blockDim.y) {
      B[j*ldb + i] = A[j*lda + i] * xval;
    }
  }
}
void diagMatrixMultMatrix(const double* x, const double* A,
    ↪ unsigned int lda, double* B, unsigned int ldb,
    ↪ unsigned int M, unsigned int N) {
  dim3 blockDim(32, 32);
  dim3 gridDim(\min((M-1) / blockDim.x + 1,32*1024),
      ↪ \min((N-1) / blockDim.y + 1,32*1024));
  diagMatrixMultMatrix_kernel<<<gridDim, blockDim>>>(x, A,
      ↪ lda, B, ldb, M, N);
}
```

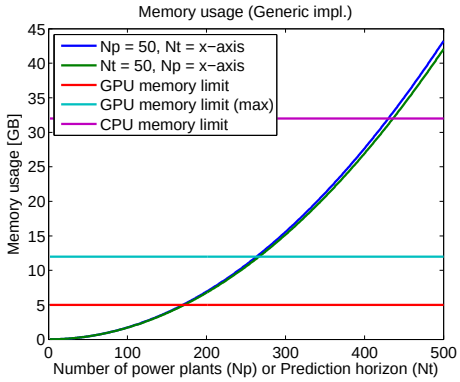


Figure 4.2: Memory usage of the generic GPU-based solver

The function `magma_dpotrf_gpu()` is the GPU-resident version, where it expects that the input is already on the GPU. It factorizes the passed positive-definite symmetric matrix in-place and replaces the input matrix with the Cholesky factorization. By doing so, it avoids requiring additional storage to hold both the normal equations matrix and the resulting Cholesky factorization.

The function `magma_dpotrs_gpu()` is the triangular solve method, which can be used to solve the system $x = L^T \backslash L \backslash x$ using the computed factor.

The function calls are shown in Listing 4.13. Note that in order for MAGMA to get the full benefit of coalesced memory access, it requires that the leading dimension of the H matrix is divisible by 16. This is done by computing $Nb = ((N-1)/16+1) \times 16$ and storing the normal equations matrix, with some padding at the end of each column, if N is not divisible by 16.

Listing 4.13: Cholesky factorization and triangular solve with MAGMA

```
// Factorize H
magma_dpotrf_gpu('L', N, H, Nb, &ret);
// Triangular solve x = L'\L\x
magma_dpotrs_gpu('L', N, 1, H, Nb, x, N, &ret);
```

4.2.3.7 Memory usage

Since CUBLAS and MAGMA are both libraries for dense computations, we must store our matrices as dense full matrices, as we did with the MATLAB

GPU implementations. Since our implementation keeps all the data on the GPU permanently to avoid memory transfers, it requires that the GPU has enough memory to store both the A matrix, which has $m \times n$ elements, and the normal equations, which has $n \times n$ elements. Additionally, we must also store the temporary result of DA, when computing the normal equations matrix $A^T DA$. This results in $(2m \times n + n \times n)$ elements required to be stored for the generic implementation.

For our test case, we have $m = 4N_t \times N_p + 2 \times N_t$ and $n = N_p \times N_t + N_t$, where N_t is the prediction horizon and N_p is the number of power plants. The memory usage is plotted on Figure 4.2 along with the memory limits of our test system. The maximum amount of memory currently available for a single GPU is 12 GB, which is also shown on the figure to show the maximum problem size capable of being solved on current state-of-the-art GPUs.

4.2.4 Results

We have run the implementation with the test case described in Chapter 3, where we keep either the prediction horizon or the number of power plants fixed to 50, and then increase the other parameter to 160, which is approximately the largest system we can solve in the C implementation, as shown on Figure 4.2.

In the following sections, we present the convergence and performance results of the generic implementations.

4.2.4.1 Convergence

The implementations were run for each problem size until they met the termination criteria in (4.12) with the tolerances $tol_p = 10^{-9}$, $tol_d = 10^{-6}$ and $tol_o = 10^{-9}$. The number of iterations required to converge is shown on Figure 4.3 on the following page. All the implementations converge in the same number of iterations and the resulting residuals are identical, implying that our implementations are numerically stable and identical.

4.2.4.2 Performance

We have run the described tests on the test machine mentioned in Section 1.4. For the MATLAB implementations, we have used sparse matrices on the CPU,

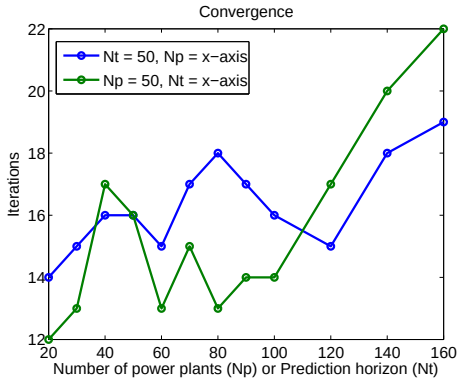


Figure 4.3: Convergence of the generic implementations

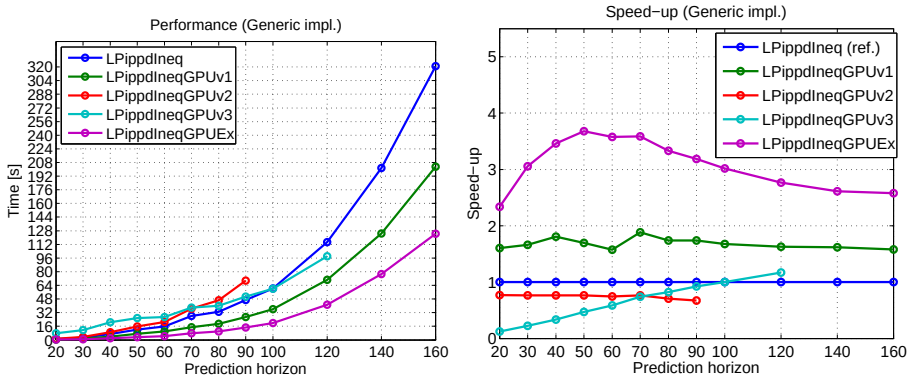
as the constraints matrix is very sparse, and we allowed MATLAB to make full use of the four cores in the test machine.

4.2.4.3 Increasing the prediction horizon

The solution time of the different implementations is shown on Figure 4.4a and a speed-up plot with the MATLAB implementation without GPU acceleration as reference is shown on Figure 4.4b.

Version 1 of the GPU-accelerated MATLAB implementation has a speed-up of approximately 1.5× to 2×. Since it only does Cholesky factorization on the GPU, there is some overhead in transfering the computed normal equations matrix to the GPU and transfering the resulting factor back, which limits the performance.

Version 2 is slower than the purely CPU-based implementation. While this version eliminates the cost of transfering the normal equations matrix to the GPU, it must still transfer the computed factor back to the CPU. Additionally, MATLAB does not have efficient support for multiplying a diagonal matrix with a general matrix on the GPU, which results in a full dense multiplication when computing the normal equations matrix. This is both highly inefficient and memory-intensive. Finally, the implementation suffers from the dense matrix-vector multiplications, as there is no sparse GPU support. The result is a slower implementation which uses a larger amount of memory than all the other implementations. This causes it to run out of memory at 100 power plants and above.



(a) Time **(b)** Speed-up

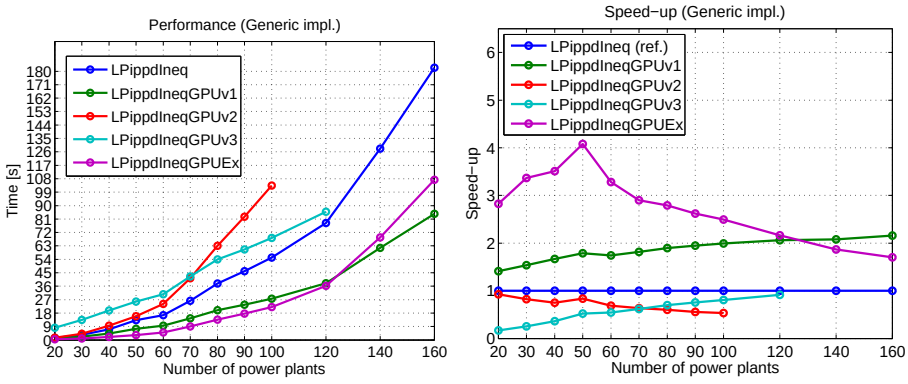
Figure 4.4: Performance of the generic implementations with variable prediction horizon.

Version 3 is much slower for smaller systems, as the cost of iterating over the rows of the diagonal matrix and scaling A row-by-row is very expensive, since it requires a separate call for each row. However, for larger systems it benefits from avoiding the dense diagonal matrix multiplication and performs slightly better than version 2 and the CPU version. It also manages to solve slightly larger systems than version 2, since it does not store D on the GPU, but still runs out of memory at 140 power plants.

The C/CUDA implementation shows the best speed-up, starting with approximately 3× to 3.5× speed-up for the smaller to medium systems, and settling slightly above 2.5× speed-up for the larger systems before running out of memory. The best speed-up is achieved for the medium systems, as the normal equations matrix reaches a sufficient size to get the best performance out of the fast Cholesky factorization on the GPU. For larger systems, the gain is slightly offset by the increased cost of multiplying with a dense constraints matrix, compared to the sparse implementation on the CPU.

4.2.4.4 Increasing the number of power plants

Increasing the number of power plants results in slightly different performance than increasing the prediction horizon. While an increase in either the prediction horizon or the number of power plants result in roughly the same increase of the size of the constraints matrix and normal equations matrix, it does make a difference in the sparsity of the constraints matrix. The solution time is



(a) Time **(b)** Speed-up

Figure 4.5: Performance of the generic implementations with variable number of power plants.

shown on Figure 4.5a and a speed-up plot with the MATLAB implementation as reference is shown on Figure 4.5b.

Version 2 and version 3 of the GPU-accelerated MATLAB implementation is again slower than the CPU implementation for the same reasons as mentioned in the previous section.

For smaller to medium sized problems, version 1 of the GPU-accelerated MATLAB implementation and the C/CUDA implementation also show similar speed-up, compared to when we increased the prediction horizon. However, for larger systems we observe a clear difference. The C/CUDA implementation drops below 2× speed-up, while the MATLAB implementation increases to above 2× speed-up and becomes the fastest implementation. This is due to the Γ part in the constraints matrix, which has dimensions $N_t \times (N_t \times N_p)$, and is the most dense part of the matrix. When the prediction horizon is increased, the number of rows of Γ increases, resulting in more dense rows. When the number of power plants is increased, the number of rows in Γ does not change, while the number of rows in the sparse constraints, such as the bounds constraints, increases. This results in more sparsity in the constraints matrix, which is beneficial for version 1 of the MATLAB GPU implementation as it, uses sparse matrix multiplication on the CPU.

Name	Implemented in	GPU usage
LPippdIneqPP	MATLAB	No
LPippdIneqPPGPUv1	MATLAB	Cholesky
LPippdIneqPPGPUv2	MATLAB	Matrix mult. + Cholesky
LPippdIneqPPGPUEx	C/CUDA	All

Table 4.2: Problem-specific solver implementations

4.3 Problem-specific implementation

In [ESJ09], the authors demonstrate that exploiting the structure in the constraints matrix, arising in a constrained optimal control problem, can substantially reduce the solution time. In this section, we specialize our implementations to our Power Plant Portfolio test case, described in Chapter 3, and extend the implementations with GPU acceleration similarly to what we did in the previous section. An overview of the problem-specific implementations described in this section is shown in Table 4.2.

4.3.1 Exploiting structure

The constraints matrix, $\bar{A}$, in our test problem from Chapter 3 is shown in (4.14).

$$\bar{A} = \begin{bmatrix} I & 0 \\ -I & 0 \\ 0 & I \\ \Psi & 0 \\ -\Psi & 0 \\ \Gamma & I \end{bmatrix} \tag{4.14}$$

The unique structure of this constraints matrix can be utilized to efficiently compute matrix-vector multiplication, as well as the normal equations matrix, which must be factorized to compute the Newton direction in the interior point method.

4.3.1.1 Matrix-vector multiplication

The matrix-vector multiplication operations with the problem-specific constraints matrix, $\bar{A}$, can be reduced to computations involving the structured sub-blocks in the constraints matrix. Let $\bar{A}$ have the structure in (4.14), then the matrix-vector and matrix-transpose-vector multiplication can be computed as shown in (4.15) and (4.16), respectively.

$$\bar{A}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix} = \begin{bmatrix} x_1 \\ -x_1 \\ x_2 \\ \Psi x_1 \\ -\Psi x_1 \\ \Gamma x_1 + x_2 \end{bmatrix} \tag{4.15}$$

$$\bar{A}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 + \Psi^T(x_4 - x_5) + \Gamma^T x_6 \\ x_3 + x_6 \end{bmatrix} \tag{4.16}$$

4.3.1.2 Constructing the normal equations matrix

The computation of the problem-specific normal equations matrix, $\bar{H} = \bar{A}^T D \bar{A}$, can be reduced to

$$\begin{bmatrix} \bar{H}_{11} & \bar{H}_{12} \\ \bar{H}_{21} & \bar{D} \end{bmatrix} \tag{4.17}$$

where

$$\begin{aligned} \bar{H}_{11} &= D_1 + D_2 + \Psi^T(D_4 + D_5)\Psi + \Gamma_u^T D_6 \Gamma_u \\ \bar{H}_{12} &= \bar{H}_{21}^T = \Gamma_u^T D_6 \\ \bar{D} &= D_3 + D_6 \end{aligned} \tag{4.18}$$

and $D_i = diag(d_i), D = [d_1, d_2, d_3, d_4, d_5, d_6]$. This follows from straight-forward matrix-matrix multiplication.

4.3.1.3 Solving Newton direction

The structure of the problem-specific normal equations matrix, $\bar{H}$, makes it possible to solve the system

$$\bar{H}\Delta x = b \tag{4.19}$$

by stating it as

$$\begin{bmatrix} \bar{H}_{11} & \bar{H}_{12} \\ \bar{H}_{12}^T & \bar{D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4.20}$$

Since $\bar{D}$ is diagonal, it is trivial to invert, and we can eliminate the second row

$$\begin{array}{lcl} \bar{H}_{12}^T\Delta x_1 + \bar{D}\Delta x_2 & = & b_2 \\ \bar{D}\Delta x_2 & = & b_2 - \bar{H}_{12}^T\Delta x_1 \\ \Delta x_2 & = & \bar{D}^{-1}(b_2 - \bar{H}_{12}^T\Delta x_1) \end{array} \tag{4.21}$$

and solve only for x_1

$$\begin{array}{lcl} \bar{H}_{11}\Delta x_1 + \bar{H}_{12}\Delta x_2 & = & b_1 \\ \bar{H}_{11}\Delta x_1 + \bar{H}_{12}(\bar{D}^{-1}(b_2 - \bar{H}_{12}^T\Delta x_1)) & = & b_1 \\ \bar{H}_{11}\Delta x_1 + \bar{H}_{12}\bar{D}^{-1}b_2 - \bar{H}_{12}\bar{D}^{-1}\bar{H}_{12}^T\Delta x_1 & = & b_1 \\ (\bar{H}_{11} - \bar{H}_{12}\bar{D}^{-1}\bar{H}_{12}^T)\Delta x_1 & = & b_1 - \bar{H}_{12}\bar{D}^{-1}b_2 \\ \hat{H}\Delta x_1 & = & \hat{b} \end{array} \tag{4.22}$$

where

$$\begin{array}{lcl} \hat{H} & = & \bar{H}_{11} - \bar{H}_{12}\bar{D}^{-1}\bar{H}_{12}^T \\ \hat{b} & = & b_1 - \bar{H}_{12}\bar{D}^{-1}b_2 \end{array} \tag{4.23}$$

This results in a smaller system to factorize as the dimension of $\hat{H}$ is $(N_t \times N_p)^2$, whereas the dimension of $\bar{H}$ is $(N_t \times N_p + N_t)^2$.

4.3.2 Plain MATLAB implementation

The MATLAB implementation of the problem-specific solver is implemented by using the generic implementation and simply replacing the normal equation solver with a problem-specific one. Instead of storing the entire A matrix and passing it to the solver, we instead pass a structure A, which contains the dimensions of A, the Ψ matrix as sparse, and the Γ matrix as dense. The Γ matrix is stored dense as this has shown to achieve the best performance. The structure also contains the function pointers `funAx` and `funAtx`, which are implemented to compute the matrix-vector multiplication, as shown in (4.15) and (4.16) respectively.

The implementation of the normal equation solver is shown in Listing 4.14. The function reuses an existing factorization from the same iteration. If the normal equations matrix has not been factorized in the iteration yet, then the sub-matrices $\bar{H}_{11}$, $\bar{H}_{12}$ and $\bar{D}$ are computed according to (4.18) and the reduced matrix $\hat{H}$ in (4.23) is factorized. Finally, x_1 is computed by solving $\hat{H}x = \hat{b}$ and x_2 is computed from x_1.

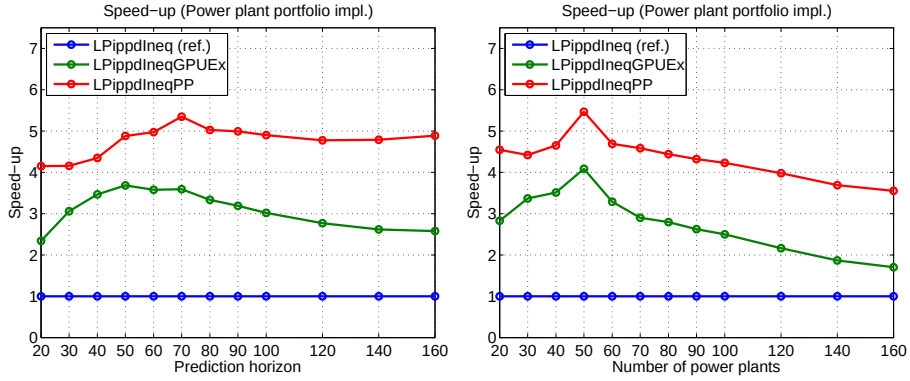
4.3.2.1 Results

We have run the same test as the one presented in Section 4.2.4.2 for the problem-specific MATLAB implementation to investigate the speed-up achieved from exploiting the structure of the constraints matrix. The speed-up when varying either the prediction horizon, or the number of power plants, is shown on Figure 4.6a and Figure 4.6b respectively.

The problem-specific solver achieves a reliable 4× to 5× speed-up compared to the generic implementation, although the performance drops slightly for systems with many power plants. As shown in (4.22), we can reduce the size of the normal equations matrix $\bar{H}$ to the reduced matrix $\hat{H}$ and only factorize this smaller matrix. The dimensions of $\hat{H}$ is $\hat{n} \times \hat{n}$, where $\hat{n} = N_t \times N_p$, while the dimensions of the full normal equation matrix $\bar{H}$ is $n \times n$, where $n = N_t \times N_p + N_t$. This means that the rows and columns we are eliminating, when exploiting the structure, are directly related to the prediction horizon. This explains the better performance when increasing the prediction horizon instead of the number of power plants.

Listing 4.14: Cholesky factorization and triangular solve with MAGMA

```
function [x, data] = funSolveNE(k, A, D, b, data)
  if (isfield(data,'k') && k == data.k && isfield(data,'L'))
    L = data.L; h11 = data.h11; h12 = data.h12;  Dbar =
        ↪ data.Dbar;
  else
    Nt = A.MPC.Nt; Np = A.MPC.Np; Psi = A.MPC.Psi;
        ↪ A.MPC.Gamma;
    [D1,D2,D3,D4,D5,D6] = ...
        splitDiagonal(D, Nt*Np, Nt*Np, Nt, Nt*Np, Nt*Np, Nt);
    h12 = Gamma' * D6;
    h11 = D1 + D2 + Psi' * (D4 + D5) * Psi + h12 * Gamma;
    Dbar = D3 + D6;
    H = h11 - h12*inv(Dbar)*h12';
    L = chol(H,'lower');
    data.k = k; data.L = L;
    data.h11 = h11; data.h12 = h12; data.Dbar = Dbar;
  end
  b1 = b(1:size(h11,1)); b2 = b(size(h11,1)+1:end);
  rhs = b1 - h12 * inv(Dbar) * b2;
  x1 = L'\(L\(rhs)); x2 = inv(h22) * (b2 - h12'*x1);
  x = [x1; x2];
end
```



(a) Increasing prediction horizon **(b)** Increasing number of power plants

Figure 4.6: Speed-up of the problem-specific MATLAB implementation compared to the generic MATLAB implementation

4.3.3 MATLAB with GPU

The problem-specific implementation in MATLAB with GPU is implemented in two versions, where version 1 uses the GPU for Cholesky factorization and version 2 uses the GPU for both matrix-multiplication and Cholesky factorization. In both versions, we have chosen to store the Γ matrix as dense, because it results in the best performance. As in the generic implementations, it still transfers the factorization back to the CPU and does the triangular solve on the CPU for reasons mentioned in Section 4.2.2.1.

The implementation of this version follows the same minor modifications, which were done to the generic version when adding GPU support, where calls to `gpuArray()` and `gather()` were added, so we will not show the code here.

4.3.4 C/CUDA implementation

The problem-specific CUDA implementation shares most of its code with the generic implementation. Like the MATLAB implementation, the key difference here is the replacement of the normal equations solver as well as the matrix-vector multiplication.

4.3.4.1 Memory usage

In the generic implementation, the constraints matrix A was stored as a dense matrix, which resulted in out-of-memory problems. In this implementation, we avoid the storage of $\bar{A}$ and only store Γ as dense instead. It is not necessary to store the Ψ matrix, as operations with the Ψ matrix can be implemented with multiple BLAS function calls. This reduces the memory usage of this implementation to $(2N_t \times (N_t \times N_p) + (n \times n))$ elements, as we only need to store Γ, $D\Gamma$ and the normal equations matrix, making it possible to solve larger systems, as shown on Figure 4.7. Thus, this will enable us to increase either N_t or N_p up to 460 in our current setup, when the other parameter is 50.

4.3.4.2 Matrix-vector multiplication

The MATLAB code for doing matrix-vector multiplication with $\bar{A}$ is shown in Listing 4.15 alongside the equivalent CUDA code in Listing 4.16.

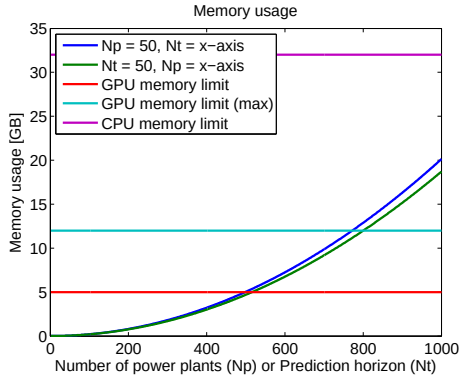


Figure 4.7: Memory usage of the problem-specific GPU-based solver

Listing 4.15: MATLAB

```
b1 = x1;
b2 = -x1;
b3 = x2;
b4 = Psi*x1
b5 = -b4;
b6 = x2;
b6 = b6+Gamma*x1;
```

Listing 4.16: CUDA

```
cublasDaxpy(Nu,1.0,x,1,b,1);
b += Nu;   // b2
cublasDaxpy(Nu,-1.0,x,1,b,1);
b += Nu; // b3
cublasDaxpy(Nt,1.0,x + Nu,1,b,1);
b += Nt; // b4
cublasDaxpy(Nu,1.0,x,1,b,1);
cublasDaxpy(Nu-Np,-1.0,x,1,b + Np,1);
cublasDaxpy(Nu,-1.0,b,1,b+Nu,1);
b += Nu*2; // b6
cublasDaxpy(Nt,1.0,x + Nu,1,b,1);
cublasDgemv('N',Nt,Nu,1.0,Gamma,
    ↪ Nt,x,1,1.0,b,1);
```

Since Γ is stored as a dense matrix, we can simply call `dgemv()` to multiply it with a vector. Multiplying the identity matrix with a vector, which is equivalent to simply copying the vector, is done with a call to `daxpy()` to add it to the resulting output vector. We do not use `dcopy()`, as we also want to multiply with -1.0 for the negative identity matrix. Note that this requires that the output vector is initialized to zero. The Ψ matrix multiplied by a vector is done with two calls to `daxpy()`, one for the main diagonal and one for the negative subdiagonal, which subtracts the previous elements.

The matrix-transpose-vector multiplication can be implemented in the same way as the matrix-vector multiplication by calling CUBLAS appropriately.

Listing 4.17: Efficient construction of problem-specific normal equations matrix in CUDA

```
// H21
mpcDGamma(Nt, Np, Gamma, d6, H + Nu, Nb);
// H11
cublasDcopy(Nu, d1plusd2, 1, H, Nb);
mpcPsiTDPsi(Nt, Np, d4plusd5, H, Nb);
cublasDgemm('T', 'N', Nu, Nu, Nt, 1.0, Gamma, Nt, H + Nu, Nb,
    ↪ 1.0, H, N);
// H22
cublasDcopy(Nt, d3plusd6, 1, H+Nu*Nb+Nu, Nb);
```

4.3.4.3 Solving the normal equations system

In this implementation, we have not exploited the fact that the normal equations can be factorized as a slightly smaller system. Instead, the full normal equations matrix, $\bar{H}$, is constructed efficiently as described in Section 4.3.1.2, and factorized. Since MAGMA's Cholesky factorization implementation works in-place, we avoid allocating additional memory to store the factorization and simply overwrite the normal equations matrix instead. Furthermore, only the lower triangle is computed as MAGMA's Cholesky factorization implementation ignores the upper-half of the matrix when computing the lower factor.

The CUDA code for the construction of $\bar{H}$ is shown in Listing 4.17. The variables `d1plusd2`, `d3plusd4` and `d3plusd6` are vectors, which are the diagonal of the diagonal matrices $diag(d1 + d2)$, $diag(d3 + d4)$, and $diag(d3 + d6)$, respectively.

The computation of $\bar{H}_{21} = D6\Gamma$ can be done with the diagonal matrix times general matrix kernel, which we described in Section 4.2.3.5. Once $\bar{H}_{21}$ is computed, we reuse the result of the computation to compute $\Gamma^T D6\Gamma = \Gamma^T \bar{H}_{21}$. The computation of $\Psi^T(D4 + D5)\Psi$ is done with four calls to `daxpy()`.

4.3.5 Results

We have run the problem-specific implementations with the test case described in Chapter 3. In the tests, we keep either the prediction horizon or the number of power plants fixed to 50 and then increase the other parameter. Since the problem-specific implementation uses much less memory, we can solve systems where the second parameter goes up to 460.

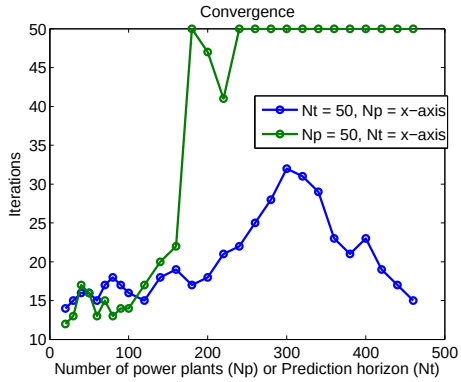


Figure 4.8: Convergence of the problem-specific implementations

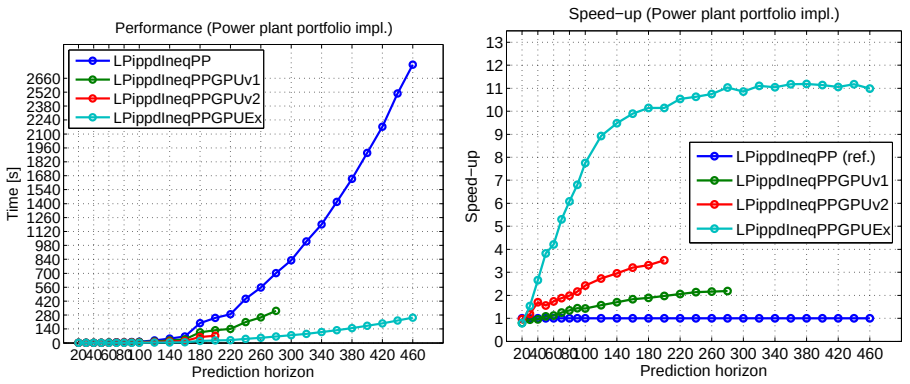
In the following sections, we present the convergence and performance results of the problem-specific implementations.

4.3.5.1 Convergence

The implementations were run for each problem size until they met the termination criteria in (4.12) with the tolerances $\text{tol}_p = 10^{-9}$, $\text{tol}_d = 10^{-6}$ and $\text{tol}_o = 10^{-9}$, or until they reach 50 iterations, which we have defined as the maximum allowed iterations. The number of iterations required to converge is shown on Figure 4.8. All the problem-specific implementations converge in the same number of iterations and the residuals are identical.

4.3.5.2 Performance

The performance tests for the problem-specific implementations were also done on the test machine mentioned in Section 1.4. In the MATLAB tests, we allow MATLAB to make full use of the four cores in the test machine.



(a) Time **(b)** Speed-up

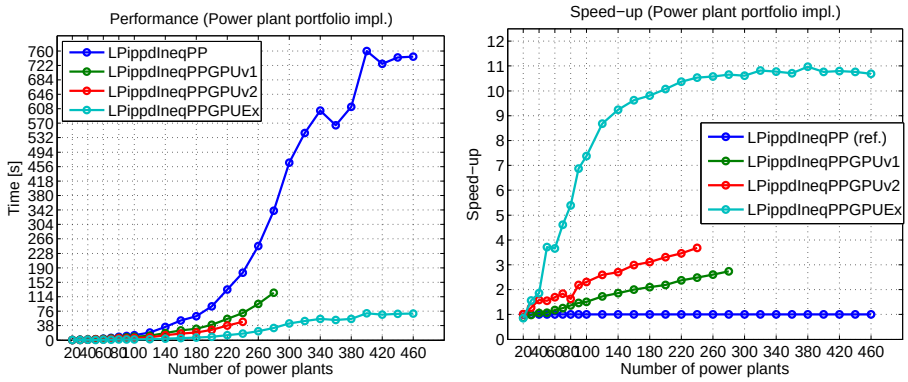
Figure 4.9: Performance of the problem-specific implementations with variable prediction horizon.

4.3.5.3 Increasing the prediction horizon

The solution time of the problem-specific implementations is shown on Figure 4.9a and a speed-up plot with the problem-specific MATLAB implementation without GPU acceleration as reference is shown on Figure 4.9b.

Version 1 of the GPU-accelerated MATLAB implementation shows almost the same speed-up as it did in the generic implementation. Both implementations benefit from the same problem-specific CPU implementation of the normal equations matrix so their performance is increased similarly. As the problem size increases, the speed-up increases slightly until it runs out of memory. Interestingly, the implementation fails to solve with a prediction horizon of 300 time steps with an out-of-memory error, yet manages to solve the larger system with 320 time steps. This appears to be an artifact of how MATLAB manages its device memory. The only memory, which is used on the device, is the normal equations matrix and the resulting Cholesky factor, as MATLAB does not perform in-place Cholesky factorization on the GPU, which should result in $2N \times N$ elements on the GPU. There is no clear explanation as to why a problem with a larger N manages to fit within device memory, while a problem with a slightly smaller N fails to do so.

Version 2 of the GPU-accelerated MATLAB implementation is much better in the problem-specific implementation with a $2\times$ speed-up for smaller systems and almost $4\times$ speed-up for larger systems. Since only the Γ matrix is stored densely on the GPU, the memory-use is less and we avoid dense multiplications with



(a) Time **(b)** Speed-up

Figure 4.10: Performance of the problem-specific implementations with variable number of power plants.

the constraints matrix, which caused the reduced performance in the generic version. However, it still runs out of memory before version 1 due to storing and computing with Γ on the GPU. This version also experiences an out-of-memory error for a smaller system of 220 time steps, while it succeeds at 240 time steps.

The C/CUDA implementation shows an impressive speed-up, starting with $2\times$ for smaller systems but rapidly increasing to $10\times$ for medium systems and up to $11\times$ for larger systems. Since this implementation is entirely on the GPU, there are no data transfers involved in the solving. Furthermore, it no longer suffers from the dense constraints matrix multiplication, which limited the generic implementation. Since the normal equations matrix is almost entirely dense, the performance benefit from using a GPU is high. This results in the best performing implementation.

4.3.5.4 Increasing the number of power plants

The solution time of the problem-specific implementations is shown on Figure 4.10a and a speed-up plot with the MATLAB implementation without GPU acceleration as reference is shown on Figure 4.10b.

The speed-up results are very similar to the previous test where we increased the prediction horizon instead. Since we have eliminated the dense constraints matrix multiplication, there is not much difference between the systems.

4.4 Conclusion

We implemented a primal-dual interior point algorithm with Mehrotra predictor-corrector for linear optimization problems in the inequality form as both a generic solver, which does not exploit the structure in the constraints matrix, as well as a problem-specific solver, which exploits the structure in our test case from Chapter 3. Both implementations were extended to include GPU acceleration for various parts of the solver in MATLAB, as well as a full GPU implementation in C. The MATLAB CPU implementations uses sparse matrices, while the GPU implementations uses dense matrices on the GPU.

The generic MATLAB implementation manages to achieve minor speed-up of 1.5× to 2×, compared to the multi-threaded MATLAB CPU implementation, by using a GPU to do the dense Cholesky factorization, as long as it still uses the sparse matrix-multiplication on the CPU. When the sparse constraints matrix is transferred to the GPU and used as a dense matrix in MATLAB, it results in worse performance than the CPU version. The best speed-up of between 1.7× to 4× is achieved by implementing the entire algorithm in C/CUDA and doing all operations on the GPU. This avoids expensive data transfers between the CPU and GPU and, even though the matrix operations with the sparse constraints matrix are done as dense, it still manages to outperform the sparse CPU implementation due to the high memory bandwidth and performance of the GPU. However, doing so incurs a large memory cost and with limited memory available on the GPU, it severely limits the maximum size of the problems possible to solve. Furthermore, extending the MATLAB version with GPUs is simple and can be done without prior knowledge of GPUs. The full C/CUDA implementation, however, requires specific implementations of CUDA kernels for operations not supported by the standard BLAS operations.

The problem-specific implementation exploited the structure of the model predictive control problem from Chapter 3 to efficiently construct the normal equations matrix and even reduced the size of the factorization. This produced roughly a 3.5× to 5.3× speed-up compared to the generic CPU implementation, performing even better than the best GPU-accelerated generic implementation. Applying GPU acceleration to the problem-specific MATLAB solver showed an additional 1.5× to 4× speed-up compared to the problem-specific MATLAB solver on the CPU. Finally, the full C implementation achieved an impressive 10× to 11× speed-up over the CPU implementation for systems with over 9000 decision variables.

Our results show that the GPU can provide a speed-up in the solution time of constrained optimization problems, when the normal equations matrix becomes dense. The GPU is exceptional at dense matrix-matrix multiplication, which is

beneficial for both the computation of the normal equations matrix and Cholesky factorization. Through the use of freely available libraries, it is possible to easily implement most of the algorithm, however some operations still require kernel implementations.

The inequality form of our test case problem resulted in a near-dense normal equations matrix, which is also why we achieved such a great speed-up using GPUs. The near-dense normal equations matrix was due to a few almost dense rows in the constraints matrix caused by the Γ sub-matrix, which causes a large dense window in the normal equations matrix $A^T DA$. A remedy to this could be to solve the optimization problem in standard form by introducing slack variables. While this increases the number of variables in the system, it results in the normal equations matrix ADA^T instead, where the sparsity is unaffected by dense rows and instead affected by dense columns. In [BFV92], Birge et al. propose choosing either the inequality form or the standard form depending on the fill-in in the normal equations matrix. We will look at solving the problem with an interior point method for linear optimization problems in standard form in Chapter 5.

Chapter 5

GPUOPT - Interior Point Method Toolbox on CPU and GPU

In Chapter 4, we solved the test case from Chapter 3 by solving the linear optimization problem in the inequality form. This form can be beneficial for model predictive control problems, as they consist mostly of inequality constraints and can easily incorporate a quadratric term in the normal equations form for quadratic optimization problems. However, in the inequality form, the normal equations matrix for our test problem resulted in an almost dense matrix. Consequently, the Cholesky factor is also dense. This was due to a few near-dense rows in the constraints matrix, caused by the Γ term, and this resulted in a high memory requirement, which limited the size of the systems we could solve. To handle larger problems, it is necessary to either reduce the storage requirements of the Cholesky factorization by maintaining sparsity, or to avoid the factorization all together by using an iterative solver.

An alternative formulation of the optimization problem is the standard form, which results in slightly different KKT conditions. The normal equations matrix in this form is affected by dense columns instead of dense rows. This makes it possible to maintain sparsity for our test case, and other similarly structured model predictive control problems.

In this chapter, we describe the development of an optimization toolbox with a primal-dual interior point method on CPU and GPU, which supports both formulations. The toolbox is called GPUOPT, which is short for GPUlab Optimization Toolbox. We design the interior point method to be modular, which separates the matrix operations and linear solver from the core interior point method. This makes it possible to easily replace the components with alternative implementations, as well as problem-specific implementations, like the one from Chapter 4.

In addition to the Mehrotra predictor-corrector algorithm used in Chapter 4, we extend the implementation of the interior point method with additional techniques to improve convergence, such as primal-dual regularization [AG99], multiple centrality correctors [Gon96], and weighted corrector directions [CG08].

5.1 Method

The primal-dual interior point method for the standard form, described in Section 2.3.3 on page 20, as well as the primal-dual interior point method for the inequality form, described in Section 4.1 on page 36, for linear optimization problems are implemented in GPUOPT. Instead of implementing the interior point method for a single form, and converting an input problem to that particular form, we use the method corresponding to the form passed by the user, as the method used can have a large influence on the performance.

While the primal-dual interior point method is implemented for both forms, we focus our description in this chapter on Mehrotra's predictor-corrector method for the standard form. The standard form of a linear optimization problem is repeated here for reference

$$\begin{array}{lll} & \text{Primal} & \\ \min\limits_{x} & \phi_p = c^T x & \\ \text{s.t.} & Ax = b & \\ & x \geq 0 & \end{array} \qquad \begin{array}{lll} & \text{Dual} & \\ \max\limits_{y,s} & \phi_d = b^T y & \\ \text{s.t.} & A^T y + s = c & \\ & s \geq 0 & \end{array} \tag{5.1}$$

In addition to the Mehrotra predictor-corrector algorithm, we extend the primal-dual interior point method with additional techniques to improve convergence, which we describe in the following sections. The techniques are described for the standard form method, but are also implemented for the inequality form.

5.1.1 Initial point

The initial point in our implementation uses Mehrotra's initial point heuristic [Meh92, NW06]. We described this heuristic for the standard form in Section 2.3.4.2 and for the inequality form in Section 4.1.2.

While our implementation does not feature any warm-start strategies, the user may optionally disable the heuristic and specify an alternative initial point in the toolbox. This can be useful when using our solver to test warm-starting strategies.

5.1.2 Step length

The step length is computed as described in Section 2.3.4.1. Per default, the implemented interior point method uses the equal step length strategy in (2.26) when taking a step, however the user can switch to the unequal primal and dual step length in (2.25). Our tests use the default equal step length strategy.

5.1.3 Termination criteria

The termination criteria implemented for the interior point method are defined as

$$\frac{\|r_p\|}{1+\|b\|} \leq tol_p \quad \text{and} \quad \frac{\|r_d\|}{1+\|c\|} \leq tol_d \quad \text{and} \quad \frac{x^T s/n}{1+|c^T x|} \leq tol_o \tag{5.2}$$

for the standard form, as described in Section 2.3.4.3, and

$$\frac{\|r_p\|}{1+\|b\|} \leq tol_p \quad \text{and} \quad \frac{\|r_d\|}{1+\|c\|} \leq tol_d \quad \text{and} \quad \frac{s^T y/m}{1+|c^T x|} \leq tol_o \tag{5.3}$$

for the inequality form, as described in Section 4.1.3.

A user callback function can also be defined, which is called at the end of every interior point iteration. In this callback function, the user can inspect the state of the interior point method, such as residuals and latest primal and dual step length, and terminate the method early. This allows for an implementation of an user-defined termination criterion.

5.1.4 Regularization

As the interior point method approaches the optimal solution, both the augmented system and the normal equations form becomes very ill-conditioned. This is due to the complementarity matrix, $\Theta = S^{-1}X$. As the iterates for the complementarity pair, (x, s), in the interior point method approaches the optimal solution, (x^*, s^*), they display a partition into the sets $\mathcal{B}$ and $\mathcal{N}$ [Gon12b, Wri97] such that

$$\begin{aligned} x_j \to x_j^* > 0 \quad \text{and} \quad s_j \to s_j^* = 0, \quad \text{for} \quad j \in \mathcal{B} \\ x_j \to x_j^* = 0 \quad \text{and} \quad s_j \to s_j^* > 0, \quad \text{for} \quad j \in \mathcal{N} \end{aligned} \tag{5.4}$$

As a consequence of this partitioning, the complementary matrix, $\Theta = S^{-1}X$, has elements which either go towards infinity or towards zero as we approach the optimal solution. Direct methods, such as Cholesky factorization, are generally not affected by this [Gon12b]. In contrast, iterative methods such as conjugate gradient are highly susceptible to ill-conditioning.

To use conjugate gradient to solve the optimization problem, Gondzio proposes a two-step method where the original problem is regularized and a partial Cholesky preconditioner is created for conjugate gradient [Gon12b]. In this section, we briefly describe the regularization, which was originally introduced in [AG99]. The preconditioner is described in Chapter 6, where we present the preconditioned conjugate gradient solver.

The regularization is done by adding a diagonal quadratic term to both the primal and dual objective function of (5.1)

$$\begin{aligned} \text{Primal} \quad \min_x \quad & \phi_p = c^T x + \frac{1}{2}(x - x_0)^T R_p (x - x_0) \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{5.5}$$

$$\begin{aligned} \text{Dual} \quad \max_y \quad & \phi_d = b^T y - \frac{1}{2}(y - y_0)^T R_d (y - y_0) \\ \text{s.t.} \quad & A^T y + z = c \\ & z \geq 0 \end{aligned} \tag{5.6}$$

The matrices R_p and R_d are diagonal regularization matrices and x_0 and y_0 are proximal terms set to the solution of the previous iteration. This results in the regularized system

$$\begin{bmatrix} -R_p & A^T & I \\ A & R_d & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} r_d \\ r_p \\ r_c \end{bmatrix} \tag{5.7}$$

which can be reduced to the following regularized augmented system form

$$\begin{bmatrix} -(\Theta^{-1} + R_p) & A^T \\ A & R_d \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \bar{r_d} - X^{-1} r_c \\ \bar{r_p} \end{bmatrix} \tag{5.8}$$

where $\Theta = S^{-1}X$. Note that by setting the proximal terms equal to the previous iteration, the right-hand side is unaffected by the regularization. The normal equation form of the regularized augmented system is

$$(A\bar{D}A^T + R_d)\Delta y = r_p + A\bar{D}(X^{-1}r_c - r_d) \tag{5.9}$$

where $\bar{D}^{-1} = (Q + \Theta^{-1} + R_p)$. If the regularization terms R_p and R_d are set to the zero matrix, then it is equivalent to solving the non-regularized problem. The use of regularization results in a perturbed system, and it is therefore important to keep the elements in R_p and R_d small. Gondzio suggests choosing R_p and R_d adaptively, where only the elements dangerously close to zero are regularized, to minimize the perturbation of the original problem. Due to time constraints, this is currently not used in our implementation and the terms are instead kept small, such as 10^{-8}.

5.1.5 Multiple centrality correctors

Multiple centrality correctors (MCC) are introduced in [Gon96], and described in [Col07] as well. It is an iterative technique to improve the centrality and achieve a larger step length. Given a search direction, $(\Delta x, \Delta y, \Delta s)$, and associated primal and dual step lengths, α_p and α_d respectively, multiple centrality correctors aim to increase the primal and dual step length for some aspiration level, $\delta \in (0, 1)$. We start by computing the following trial point

$$\bar{x} = x + (\alpha_p + \delta)\Delta x \tag{5.10}$$

$$\bar{s} = s + (\alpha_d + \delta)\Delta s \tag{5.11}$$

and the corresponding complementarity products

$$\bar{v} = \bar{X}\bar{S}e \tag{5.12}$$

where $\bar{X} = diag(\bar{x})$ and $\bar{S} = diag(\bar{s})$. Then we define a target for the Newton direction, where we attempt to bring the complementarity elements within a symmetric neighbourhood of the central path, $\mathcal{N}_s(\gamma)$, for some $\gamma \in (0, 1)$, such that

$$\gamma\mu \leq \bar{v}_i \leq \gamma^{-1}\mu \tag{5.13}$$

The target, t, is then defined as

$$t_i = \begin{cases} \gamma\mu - \bar{v}_i & \text{if} \quad \bar{v}_i \leq \gamma\mu \\ \gamma^{-1}\mu - \bar{v}_i & \text{if} \quad \bar{v}_i \geq \gamma^{-1}\mu \\ 0 & \text{otherwise} \end{cases} \tag{5.14}$$

and the corrector is computed by solving

$$\begin{bmatrix} -R_p & A^T & I \\ A & R_d & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x_m \\ \Delta y_m \\ \Delta s_m \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ t \end{bmatrix} \tag{5.15}$$

for the regularized system. Once the direction is computed from solving (5.15), it is added to the original direction to compute the new direction

$$(\Delta y_p, \Delta y_p, \Delta s_p) = (\Delta x, \Delta y, \Delta s) + (\Delta x_m, \Delta y_m, \Delta y_m) \tag{5.16}$$

Subsequently, the primal and dual step lengths, $\bar{\alpha}_p$ and $\bar{\alpha}_d$ respectively, are computed for the new direction. In our implementation, we evaluate the primal and dual separately when selecting directions, and only accept the new corrector if it actually increases the step length, such that

$$(\alpha_p, \Delta x) = \begin{cases} (\bar{\alpha}_p, \Delta x_p) & \text{if} \quad \bar{\alpha}_p > \alpha_p \\ (\alpha_p, \Delta x) & \text{otherwise} \end{cases} \tag{5.17}$$

$$(\alpha_d, \Delta y, \Delta s) = \begin{cases} (\bar{\alpha}_d, \Delta y_p, \Delta s_p) & \text{if} \quad \bar{\alpha}_d > \alpha_d \\ (\alpha_d, \Delta y, \Delta s) & \text{otherwise} \end{cases} \tag{5.18}$$

Multiple centrality correctors can be used iteratively by computing a new corrector from the computed direction of a previous multiple centrality corrector iteration. The optimal number of centrality correctors to apply is problem-dependent, and also heavily depends on the linear solver used to compute the Newton direction.

Computing the multiple centrality corrector requires computing a new Newton direction. If a direct method is used, the factorization can be reused and the corrector is cheap to compute compared to the cost of the factorization. In contrast, if an iterative method is used, it is expensive to compute the corrector and the cost of computing the corrector may be greater than the benefit of using a multiple centrality corrector. For iterative methods, it is therefore best to keep the number of centrality correctors to a minimum or even avoid multiple centrality correctors entirely depending on the problem.

In our implementation, we let the user define the maximum number of centrality correctors to apply per interior point iteration, and also terminate early if a centrality corrector does not improve the iteration step length, $\alpha_k = \min(\alpha_p, \alpha_d)$,

Listing 5.1: Multiple centrality correctors

```
Given iterate (x, y, s), search direction (Δx, Δy, Δs), primal and
    ↪ dual step length (α_p, α_d), and maximum number of
    ↪ correctors (k)
do
  Compute iteration step length α_k = min(α_p, α_d)
  Compute (Δx_m, Δy_m, Δs_m) by solving (5.15)
  Compute (Δx_p, Δy_p, Δs_p) according to (5.16)
  Compute step length ᾱ_p and ᾱ_d such that
    x + ᾱ_p Δx_p > 0 and s + ᾱ_d Δs_p > 0.
  Set (α_p, Δx) and (α_d, Δy_p, Δs_p) according to (5.17) and (5.18).
while (number of iterations < k and min(α_p, α_d) ≥ 0.1α_k)
```

by a fraction of its aspiration level, δ. The full algorithm is summarized in Listing 5.1.

5.1.6 Weighted corrector directions

Recognizing the fact that the correctors in interior point methods do not always increase the step length, Colombo et al. introduce weighted corrector directions [CG08]. As the name implies, weighted corrector directions apply a weight to the corrector direction, independent of the step length, instead of applying the full corrector to the Newton direction. Given a search direction, $(\Delta x, \Delta y, \Delta s)$, and a corrector, $(\Delta x_c, \Delta y_c, \Delta s_c)$, the new search direction is determined by computing

$$\Delta_x = \Delta x_a + \omega_p \Delta x_c \tag{5.19}$$

$$(\Delta y, \Delta s) = (\Delta y_a, \Delta s_a) + \omega_d (\Delta y_c, \Delta s_c) \tag{5.20}$$

where $\omega_p \in (0, 1]$ and $\omega_d \in (0, 1]$. This allows different weights in the primal and dual direction as also proposed in [CG08].

The weights, ω_p and ω_d, are found by using a simple line search algorithm. The search interval is defined as $[\alpha_p \alpha_d, 1]$ and a line search of k steps is done to find the optimal weight. Weighted corrector directions can be applied to the corrector term in Mehrotra's predictor-corrector algorithm, as well as the corrector terms in the multiple centrality correctors algorithm. This is done in our implementation and the user may specify the maximum number of steps to use in the line search. If the number of steps is set to 1, then it is equivalent to disabling weighted corrector directions and taking a full corrector step. Evaluating

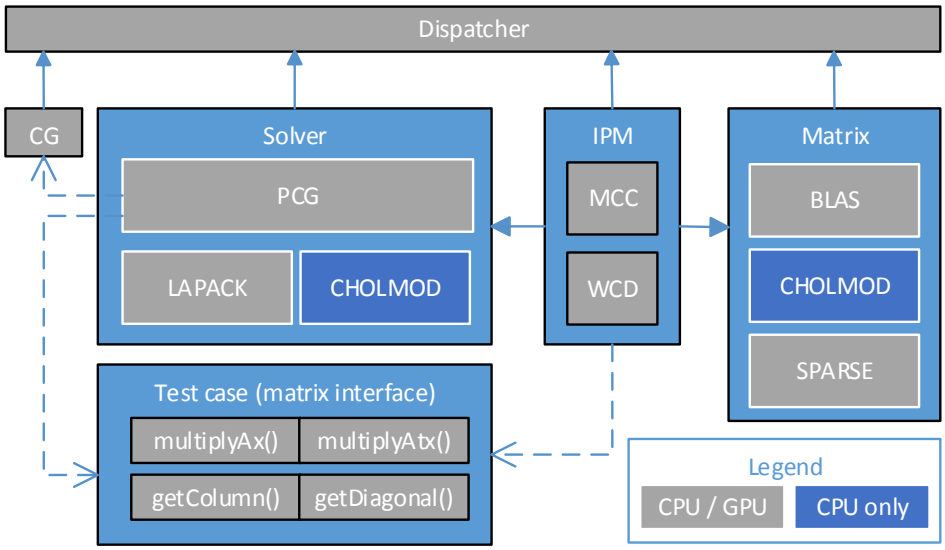


Figure 5.1: Component overview of toolbox

a weight for weighted corrector directions is inexpensive, as it does not require solving the KKT system to compute a new Newton direction.

5.2 Implementation overview

The implementation of our interior point method is split into multiple components, which are connected through function pointers. This makes it possible to easily replace a particular component with an alternative implementation of it. We briefly introduce the components here, and then go in further detail in the following sections. Figure 5.1 shows an overview of the components, along with the implemented versions of each.

The IPM component is the main interior point method component, which implements the primal-dual interior point method described in Section 5.1.

The Matrix component is the matrix-vector multiplication component used by the IPM component to do all the matrix-vector multiplications, e.g. when

residual vectors are computed. We have implemented the matrix-vector operations with dense BLAS operations, as in Chapter 4, as well as a general sparse matrix-vector. The sparse implementation is done with CUSPARSE for the GPU and with both CHOLMOD [Dav] and standard C on the CPU.

The Solver component consists of a number of different linear system solvers, which are used by the IPM component to compute the Newton direction in (2.29). It includes a dense factorization solver based on LAPACK, a preconditioned conjugate gradient solver for the normal equations form and a sparse Cholesky factorization solver using CHOLMOD. CHOLMOD is a supernodal sparse Cholesky factorization implementation for the CPU by Tim Davis [Dav].

The CG component is an implementation of the preconditioned conjugate gradient algorithm, which is used by the PCG-based linear solver. This component is implemented entirely independent of the interior point method and can be used as a generic preconditioned conjugate gradient solver on either CPU or GPU. It is described in detail in Chapter 6.

The Dispatcher component is used by all the other components to avoid implementing the algorithms twice, both for CPU and GPU. It uses a function table to dispatch the numerical operations and memory allocation to either the CPU or GPU.

The Test case component is an implementation of the matrix component interface for our test case from Chapter 3. It includes both matrix-vector multiplications, satisfying the interface for the Matrix component, as well as additional functions required by the preconditioned conjugate gradient method. The problem-specific functions have been implemented for both the inequality form and the standard form of the problem. It serves as an example of how the toolbox may be used with a problem-specific implementation. This component is described in Chapter 6.

5.3 Dispatcher

The dispatcher is used to delegate numerical operations to either the CPU or the GPU. This allows us to unify most of the code for our algorithms. Instead

Listing 5.2: Partial vtable

```
typedef struct dispatch {
  dispatchVariant_t variant;
  ...
  void (*daxpy)(const int N, const double alpha, const double
      ↪ *X, const int incX, double *Y, const int incY);
  double (*dnrm2)(const int N, const double *X, const int
      ↪ incX);
  ...
} dispatch_t;
```

of having separate implementations for both the CPU and GPU, the dispatcher is implemented as a struct, containing function pointers to each of BLAS and LAPACK functions needed for the implementation of the interior point method, as well as additional functions. The additional functions are functions such as the minimum function described in Section 4.2.3.3, and memory handling functions, such as `malloc()`. The struct is similar to a virtual method table (vtable) used in C++ for virtual functions in classes. Listing 5.2 shows a partial view of the vtable, where `daxpy` and `dnrm2` are function pointers.

The vtable is initialized with a function call, which specifies the desired variant or target. The interface is shown in Listing 5.3. This function call initializes function pointers in `vtable` according to the passed variant. In our current implementation, we have implemented the general variants `DISPATCH_VARIANT_HOST` and `DISPATCH_VARIANT_DEVICE`.

Listing 5.3: Initialization of dispatch table

```
dispatchInitialize(dispatch_t* vtable, dispatchVariant_t
    ↪ variant)
```

The variant `DISPATCH_VARIANT_HOST` is our general CPU implementation. It uses CBLAS for BLAS operations, some manually implemented functions for extended operations, such as element-wise vector operations, and memory is allocated on the CPU. When using this variant, it is possible to link with any CBLAS compliant library and the interior point method uses this for its computations.

The variant `DISPATCH_VARIANT_DEVICE` is our general GPU implementation for CUDA. It uses CUBLAS for BLAS operations and memory is allocated on the GPU. The non-BLAS operations, such as element-wise vector operations, are implemented as separate CUDA kernels as we did in Chapter 4.

Listing 5.4: Example of using dispatcher. A vector with m elements is allocated on the GPU, every element is set to 42 and the Euclidean norm is computed.

```
dispatchInitialize(&vtable, DISPATCH_VARIANT_DEVICE);
double* D = vtable.calloc(m, sizeof(double));
vtable.dsetx(m, 42.0, D, 1);
double norm = vtable.dnrm2(m, D, 1);
vtable.free(D);
```

The use of the dispatcher in our implementation unifies most of the code regardless of whether it is on the CPU or the GPU. It also simplifies extending the implementation to different targets, such as such a potential future implementation in OpenCL, by implementing an additional variant. An example of the use of the dispatcher is shown in Listing 5.4.

Not all operations needed for the implementation of the components are placed in the dispatcher, as some are very specific for the individual component. For those operations, the components may inspect the `variant` variable in the vtable to determine the variant and handle the operation accordingly. We show an example of this in the implementation of multiple centrality correctors in Section 5.4.1.

5.4 Interior point method

The interior point method component is the core component, which implements the primal-dual interior point algorithm described in Section 5.1. The component is designed around a struct called `ipmContext_t`. The context structure holds all necessary data needed by the solver to solve an linear optimization problem. The context is initialized by calling `ipmCreateContext()`, which has the prototype shown in Listing 5.5.

Listing 5.5: IPM interface to create context

```
void ipmCreateContext(ipmContext_t* context, dispatch_t*
    ↪ vtable, ipmForm_t form, int m, int n, void
    ↪ (*multiplyQx)(...), const void* quadraticUserData,
    ↪ double* c, void (*multiplyAx)(...), void
    ↪ (*multiplyAtx)(...), const void* constraintUserData)
```

The function takes a pointer to a dispatcher, which is then used throughout the interior point method. The `form` parameter is either `IPM_FORM_STANDARD` or `IPM_FORM_INEQUALITY`, to indicate whether the optimization problem is stated in the standard form or in the inequality form, respectively.

Other parameters of interest are `multiplyAx`, `multiplyAtx` and `userData`. The first two are function pointers, which must point to user-defined functions, that compute the matrix-vector and matrix-transpose-vector product with the constraints matrix A, respectively. This allows matrix-free use of the interior point method, similar to the MATLAB implementation in Chapter 4 and to [Gon12b]. The parameter `constraintUserData` is saved in the context and passed to the user-defined functions when they are called by the interior point method, which can contain any necessary data that the user-defined functions need to compute the matrix-vector multiplications.

The parameters `multiplyQx` and `quadraticUserData` are used in a similar way to handle the quadratic term of the objective function for quadratic problems. We do not deal with quadratic problems in this work, though.

The use of the matrix-free interior point method limits the storage requirements of the algorithm. This makes it possible to solve very large problems, as well as simplifies the implementation of efficient problem-specific solvers, similar to the solvers in Chapter 4. We demonstrate this in detail in Section 6.3, where we discuss the implementation of the problem-specific functions for our test case from Chapter 3. Alternatively, we have implemented a couple of general matrix-vector functions, which we cover in Section 5.5. These can be used if problem-specific functions are not available.

In addition to computing the matrix-vector product, the interior point method also requires a function to compute the Newton direction. The matrix-vector functions make it possible for the interior point method to compute the residuals and other values, which require the constraints matrix. However, it must also be able to solve (5.7) to compute the Newton directions for multiple different right-hand sides. To maintain a matrix-free implementation, and decouple the linear solver from the interior point algorithm, this is also done through a function pointer. The linear solver is set by calling `ipmSetSolver()`, which has the prototype shown in Listing 5.6.

Listing 5.6: IPM interface to set linear solver

```
void ipmSetSolver(ipmContext_t* context,
    void (*solveNormalEquations)(...),
    void (*solveAugmentedSystem)(...), const void* userData);
```

The parameters `solveNormalEquations()` and `solveAugmentedSystem()` are function pointers, which must compute the Newton direction by solving the normal equations form or the augmented system form, respectively. In this work, we focus on solvers for the normal equations form, however, the ability to assign a solver for the augmented system is important for some problems. For instance, the normal equations form for quadratic problems in the standard form requires inverting the quadratic term, which can be expensive depending on its structure. In such a case, it can be more efficient to solve the augmented system instead [Gon12b, AGMX96].

Similar to the MATLAB implementation in Chapter 4, we use this feature to define a linear solver, independently of the interior point method, to implement different linear solvers, which we describe in detail in Section 5.6.

Figure 5.2 shows an example of the interaction between the IPM component, the matrix component, and the solver component when solving a linear op-

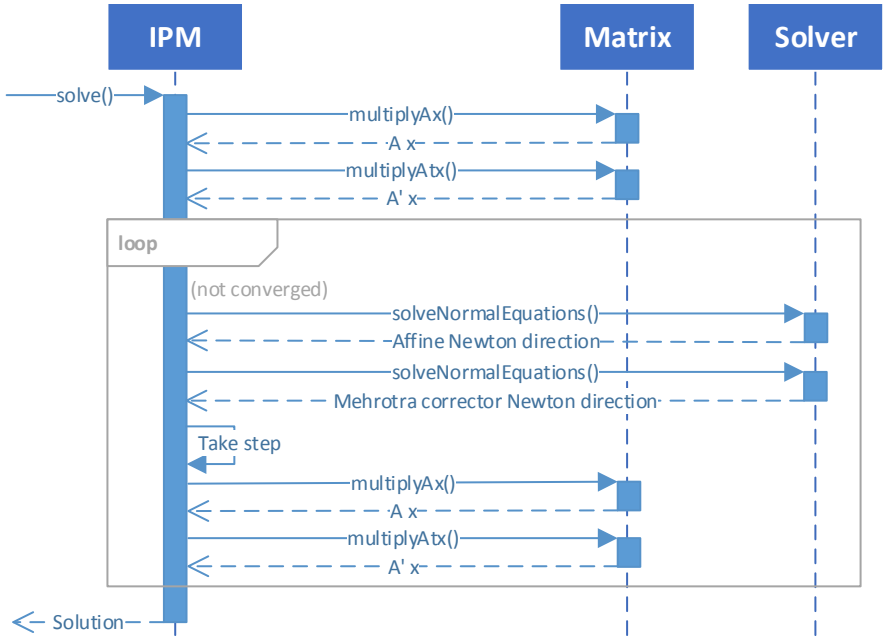


Figure 5.2: Sequence diagram showing component interaction for `ipmSolve()` when solving a linear optimization problem without multiple centrality correctors.

timization problem with the normal equations solver, without using multiple centrality correctors. The algorithm uses the matrix-vector component to compute the residuals. The linear solver component is then used to compute the affine Newton direction, by passing it the computed residual vectors as the right-hand side. The solver component is then called again to compute the corrector direction, a step is taken and the matrix component is used to compute the new residual vectors. This loops until the interior point method converges or the maximum number of iterations are reached. Note that the diagram only shows component interactions and not the internal steps in the interior point method, such as computing the centering parameter and step lengths.

In the following sections, we describe some of the implementation details of the interior point method.

5.4.1 Multiple centrality correctors

Multiple centrality correctors (MCC) are implemented as described in Section 5.1.5. The implementation of MCC requires one atypical operation, which cannot be implemented with BLAS-like operations. This is the computation of the target, which defines its elements according to the conditions in (5.14).

For the CPU implementation, this is a straight-forward for-loop, which iterates over the elements in t, and sets the value with a simple if-statement. For the GPU, it is necessary to implement a kernel to compute the target vector. We do not show the entire kernel here, as it adopts the same structure as all the other vector operation kernels, where a single element is computed independently of the other elements, such as the element-wise vector multiplication kernel shown in Listing 4.9 on page 47. However the computation of the element value is interesting to show as it differs from the CPU.

Due to the SIMT architecture of the GPU, using an if-condition to assign the target value is inefficient, as the branching results in the GPU executing the two different branches sequentially. Instead, we can use a trick in C where the result of a conditional statement is evaluated to one if it is true, and zero if it is false. Using this, we can assign the target element values as shown in Listing 5.7, which shows the inner part of the kernel loop.

Listing 5.7: Computing target element values in multiple centrality correctors on the GPU

```
double val = (x[idx] + alphaPT * dx[idx]) * (s[idx] +
    ↪ alphaDT * ds[idx]);
t[idx] = (val <= limitLower) * (limitLower - val) + (val >=
    ↪ limitUpper) * (limitUpper - val);
```

The values `limitLower` and `limitUpper` are passed to the kernel and are the lower and upper bounds of the symmetric neighbourhood, which are $\gamma\mu$ and $\gamma^{-1}\mu$ as shown in (5.13), respectively. The `val` variable is computed to be the corresponding element in the trial points complementarity products, $\bar{v}$. The assignment of the target element computes the conditional statement and value for both the lower bound and the upper bound, and simply multiplies the value with the corresponding conditional statement.

If a condition is true, then it is evaluated to one and the target element gets the associated value. If both conditions are false, then both values are multiplied with zero and the target value is set to zero. While this trick uses more floating point operations than a if-branch, this is efficient on the GPU, where the if-branch is expensive due to sequential execution when branching, while the floating point operations throughput is very high.

On an additional implementation note, we define the defaults $\gamma = 0.1$ and $\delta = 0.3$ as used in [CG08], and use these values for our tests. They can be manually set by the user through the IPM context.

5.4.2 Weighted corrector directions

The weighted corrector directions are straight-forward to implement using the dispatcher. The code for it is shown in Listing 5.8. It consists of a for-loop, which iterates through the line search. Each iteration computes the search direction for a weight, and computes the associated primal and dual step lengths. On the GPU, we use the same kernel we described in Section 4.2.3.4 to do this, while on the CPU we use a simple for-loop. If a better primal or dual step length is found, then it is stored along with the optimal weight.

5.5 Matrix component

The matrix component consists of functions used to handle the constraints matrix, A. Each implementation must provide these two functions with the interface shown in Listing 5.9

Listing 5.9: Matrix-vector component interface

```
void multiplyAx(const int m, const int n, const double *x,
    ↪ double *Ax, const void* userData);
void multiplyAtx(const int m, const int n, const double* x,
    ↪ double* Atx, const void* userData), const void*
    ↪ userData);
```

In this section, we describe the general matrix-vector multiplication functions implemented in the toolbox. These functions are useful if a problem-specific implementation is not available, as they only require the matrix is loaded in memory in the appropriate format. In Section 6.3, we show a problem-specific implementation for our test case in Chapter 3.

Listing 5.8: Weighted corrector directions implementation

```
for (int i = 0; i < steps; i++) {
  double omega = lower+step*i;
  // Take step
  vtable->dcopy(n,dxp,1,dx,1);
  vtable->dcopy(n,dsp,1,ds,1);
  vtable->daxpy(n,omega,dxc,1,dx,1);
  vtable->daxpy(n,omega,dsc,1,ds,1);
  // Compute step length
  ipmComputeSteplength(context,x,s,dx,ds,&alphaPhat,&alphaDhat);
  // Accept better omegas
  if (alphaPhat >= *alphaP) {
    *alphaP = alphaPhat;
    *omegaP = lower + step * i;
  }
  if (alphaDhat >= *alphaD) {
    *alphaD = alphaDhat;
    *omegaD = lower + step * i;
  }
}
```

5.5.1 BLAS

The most basic matrix-vector functions we have implemented are `ipmMultiplyAxCBLAS()` and `ipmMultiplyAtxCBLAS()`, which simply use CBLAS to multiple a dense column-wise matrix with a vector, as well as `ipmMultiplyAxCUBLAS()` and `ipmMultiplyAtxCUBLAS()`, which do the same but with CUBLAS [NVIa]. They expect the `userData` parameter to be a pointer to the constraints matrix stored as a column-wise dense matrix on CPU for the CBLAS variant and on the GPU for the CUBLAS variant. As these functions require the dense storage of the constraints matrix, they are very memory-inefficient as we saw in Chapter 4.

5.5.2 CHOLMOD

CHOLMOD is part of SuiteSparse by Tim Davis [Dav]. It is a supernodal sparse Cholesky factorization library. It is very efficiently implemented and includes various orderings such approximate minimum degree (AMD) ordering and column approximate minimum degree (COLAMD) ordering, which helps perserve sparsity and reduces the storage and computational requirement of the factorization. CHOLMOD is a CPU implementation, although since version 2.0 it has included some GPU support, where the dense sub-blocks in the sparse factorization are multiplied using CUBLAS. Unfortunately, we were not able to make this work for our tests.

As we use this library for one of our linear solvers, which we describe in Section 5.6.2, we have also implemented the two matrix-vector functions `ipmSolverCHOLMODMultiplyAx` and `ipmSolverCHOLMODMultiplyAtx()`. These functions use CHOLMOD to compute the matrix-vector and matrix-transpose-vector product between a CHOLMOD loaded matrix and a dense vector. The user data must be set to the same user data used by the solver component.

5.5.3 General sparse formats

In the following, we briefly introduce the implementations of matrix-vector multiplication in general sparse formats available in GPUOPT. The formats used are based on the available formats in the CUSPARSE library [NVIb], as the GPU implementations uses this library to do the matrix-vector multiplications. For further details on the formats and their GPU implementation, we refer to [BG08, BG09].

5.5.3.1 CSR

Compressed Sparse Row (CSR) format, also known as Compressed Row Sparse (CRS), is a general sparse matrix format. Given a matrix of size $m \times n$ with nnz non-zeros, the matrix is stored using three vectors, $vals$, $rows$ and $cols$. The vector $vals$ is a dense vector of length nnz containing all the non-zero elements in the matrix in row-wise order. The $cols$ vector is an index vector of length nnz which contains the column index of the corresponding value in $vals$. The $rows$ vector of length $n+1$ is an index vector which contains the index of where a corresponding row starts in $vals$ and $cols$. The last index in $cols$ is set to nnz.

The format is very compact and allows for straight-forward access to the rows in the matrix, however accessing the columns requires accessing all the non-zero elements in the matrix.

We have implemented the matrix-vector multiplication functions for this format on both the CPU and GPU. The CPU implementation is implemented in C, while the GPU implementation uses the CUSPARSE function `cusparseDcsrmv()`.

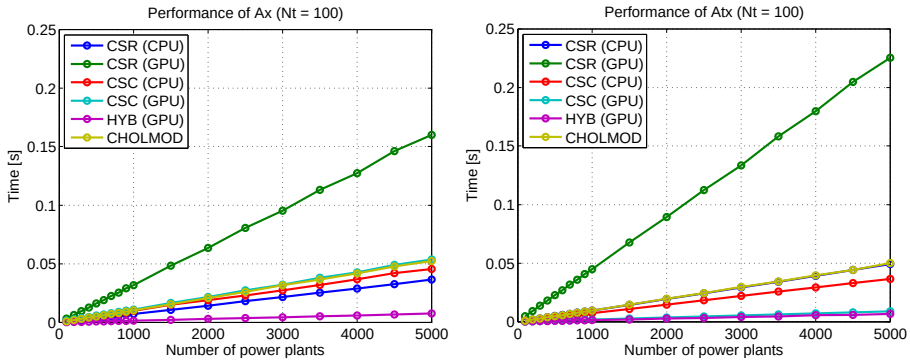
5.5.3.2 CSC

Compressed Sparse Column (CSC) format is similar to CSR, except the matrix is stored column-wise instead of row-wise and the index pointers are switched. A matrix stored in CSC is equivalent to storing the transpose of a matrix in CSR. This allows for straight-forward access to the columns in the matrix instead of the rows.

5.5.3.3 HYB

The Hybrid (HYB) format is only implemented for the GPU by using the CUSPARSE library. It combines two different sparse formats, ELLPACK (ELL) format and Coordinate (COO) format, to store the matrix for efficient matrix-vector multiplication. it is described in further detail in [BG08, BG09].

The matrix-vector multiplication is done with `cusparseDhybmv()`, however this currently only supports non-transpose matrix-vector multiplication. To compute the matrix-transpose-vector multiplication as well, we store the matrix both as non-transpose and as transpose.



(a) Ax **(b)** $A^T x$

Figure 5.3: Performance of the matrix-vector multiplication functions.

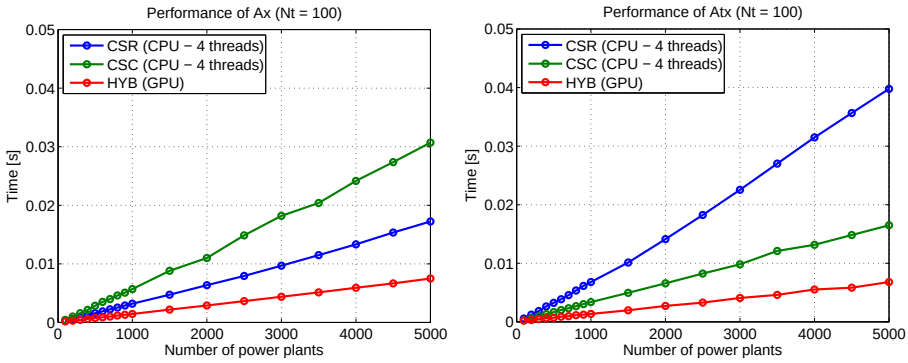
5.5.4 Performance

We have tested the performance of the generic matrix-vector multiplication implementations with our test case from Chapter 3 in the standard form. The BLAS matrix-vector implementation is not included, as it cannot handle large problems due to memory requirements. Like the tests in Chapter 4, the test was run on the test machine from Section 1.4. The performance of the matrix-vector product and the matrix-transpose-vector product are shown on Figure 5.3a and 5.3b respectively. The CPU versions were run sequentially.

The performance for the all sparse formats on the CPU are very similar. As all of them are running sequentially, there are no issues with parallelization which may give one format an advantage over another.

The GPU implementations show a large difference in performance depending on format. Storing the matrix as CSR gives very bad performance, while storing the matrix in CSC gives equivalent performance to the sequential CPU for matrix-vector and about 4× to 5× speed-up for matrix-transpose-vector compared to the sequential CPU versions. As both the CSR and CSC implementations use the same matrix-vector functions, `cusparseDcsrmv()`, then this simply seems to be a result of the combination of the structure of our constraints matrix and the parallel implementation of the CSR matrix-vector kernel implemented in CUSPARSE.

The best performing implementation is with the HYB format on the GPU. For both the matrix-vector and the matrix-transpose vector multiplication, it achieves 4× to 5× speed-up compared to the sequential CPU versions.



(a) Ax **(b)** $A^T x$

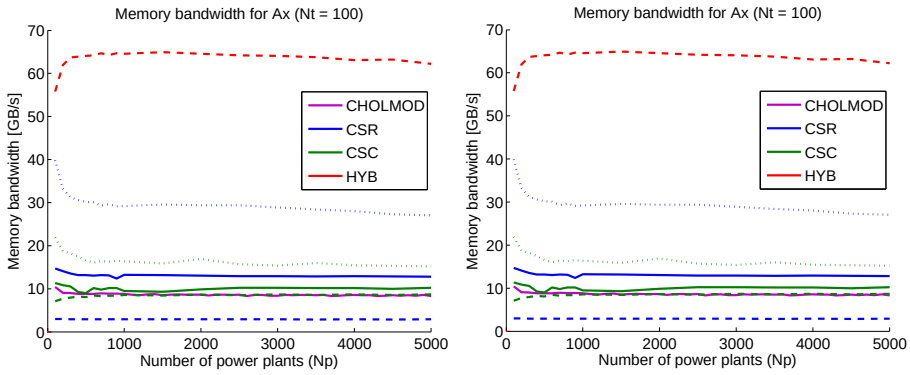
Figure 5.4: Performance of the matrix-vector functions with multi-threaded CPU implementations.

5.5.4.1 Parallel

While the HYB format on the GPU is many times better than CSR and CSC on a GPU for our test problem, the comparison to a sequential CPU is rather unfair as all new CPUs have multiple cores. We have implemented some basic multi-threading for the CSR and CSC CPU implementations by using OpenMP and run the test with all four cores in our test machine to give a better comparison between the GPU and the CPU. The results are shown on Figure 5.4a and Figure 5.4b. The performance is much closer now between the CPU and GPU. The HYB format on the GPU still performs the best, but the difference is much smaller as it is only about 2× times faster now than CSR for matrix-vector and CSC for matrix-transpose-vector.

5.5.4.2 Memory bandwidth

Sparse matrix-vector multiplication is a memory-bound operation [BG08, BG09], which means that the performance is limited by the memory bandwidth instead of the performance of the floating point operations. Figure 5.5 shows the effective memory bandwidth of the different implementations, as defined in [BG08]. The effective bandwidth is the achieved bandwidth of the computation in the absense of a cache, such that each element must be loaded when it is used for computation. This is computed by calculating the number of non-zeros elements in the constraints matrix. The calculated memory bandwidth only includes the actual values loaded and store, and does not include the overhead required for



(a) Ax **(b)** $A^T x$

Figure 5.5: Memory bandwidth of the matrix-vector multiplication functions, when including only the memory access to the values. Solid lines are for the CPU, dotted lines are for multi-threaded CPU (4 cores), and dashed lines are for the GPU.

each format, such as index vectors.

The memory bandwidth of the test machine is approximately 42 GB/s for the CPU and 208 GB/s for the GPU, theoretically. The multi-threaded CPU implementation of CSR for matrix-vector multiplication and CSC for matrix-transpose-multiplication achives a bandwidth utilization of 30 GB/s, which fully utilizes the CPU bandwidth as the remaining bandwidth is spent on the indexing vectors. Even if the CPU had more cores, the performance would not improve due to the limitation of the memory bandwidth. The GPUs high memory bandwidth allows the HYB kernel to achieve a bandwidth utilization of over 60 GB/s, resulting in twice the performance of the multi-core CPU. This is still rather low compared to the GPUs theoretical memory bandwidth. In Section 6.3 we present a problem-specific implementation, which improves the performance by eliminating the overhead spent on indexing vectors.

5.6 Linear solvers

The linear solver component is our collection of functions which implement either the normal equations form solver interface or the augmented systems solver interface. The interfaces are shown in Listing 5.10. In this work, we have only implemented solvers for the normal equations form. The implementation

of this function must solve the system

$$(ADA^T + E)x = b \tag{5.21}$$

for the standard form and

$$(A^T DA + E)x = b \tag{5.22}$$

for the inequality form. The system it should solve is indicated by the `form` parameter.

The solver can be used by the interior point method to solve all the different Newton directions in the algorithm by passing different right-hand sides to the function. For instance, to solve the Newton direction for a regularized linear optimization problem in the standard form, the interior point method calls the function with $E = R_d$, $D = (\Theta^{-1} + R_p)^{-1}$ and an appropriately computed b corresponding to the right-hand side of the system it is solving. For the affine direction, this would be $b = r_p + A\bar{D}(X^{-1}r_c - r_d)$ as shown in (5.9).

There are two steps involved when implementing a linear solver component for our interior point method implementation. The first step is to implement either of the solver interfaces, shown in Listing 5.10. The second step is the creation of some solver data, specific to the solver implementation, which is passed to the solver function through the `data` parameter. The solver data must contain all the data the solver needs to solve the linear system passed to it.

In the following sections, we describe the factorization-based implementations in GPUOPT. In Chapter 6, we extend GPUOPT with an iterative matrix-free linear solver based on conjugate gradient.

Listing 5.10: IPM linear solver interface

```
void (*solveNormalEquations)(const int k, ipmForm_t form,
    ↪ const double* D, const double E, const double* b,
    ↪ double* x, const void* data);
void (*solveAugmentedSystem)(const int k, ipmForm_t form,
    ↪ const double* D, const double Rp, const double Rd,
    ↪ const double* f, const double* g, double* dx, double*
    ↪ dy, const void* data);
```

Listing 5.11: Solver data for LAPACK-based solver

```
typedef struct {
  int m, n;              ///< Dimension of A.
  double* A;             ///< Dense column-wise A matrix.
  double* tmp;           ///< Temporary memory
  dispatch_t* vtable;    ///< Dispatcher
  int iteration;         ///< IPM iteration L was computed in.
  double* L;             ///< Factorization
} ipmSolverLapackData_t;
```

5.6.1 LAPACK

The simplest solver implementation is a dense factorization of the normal equations matrix using LAPACK, as we did in Chapter 4. This requires storing the constraints matrix as well as the factorization as dense, which is extremely memory heavy. However, it is the simplest solver to implement, and serves as a good introduction to the required steps when implementing a new solver.

5.6.1.1 Solver data

The solver data for this solver must contain the constraints matrix, in order to form the normal equations matrix, as well as the allocated memory for storing the factorization and temporary data. Finally, it must also contain a pointer to the dispatcher used in the interior point method, as this allows it to work with both CPU and GPU. The full solver data structure is shown in Listing 5.11.

To make it easy for a user to use the solver with the interior point method, two functions have been implemented to create and destroy the solver data structure. The function `ipmNESolverLAPACKCreate()` allocates the necessary memory in the passed data structure and assigns the fields in the data structure appropriately, while the destroy function handles clean-up of the solver data. The function prototypes for these two functions are shown in Listing 5.12.

Listing 5.12: Solver data functions for LAPACK-based solver

```
void ipmSolverLAPACKCreate(ipmSolverLAPACK_t* solverData,
    ↪ dispatch_t* vtable, int m, int n, double* A);
void ipmSolverLAPACKDestroy(ipmSolverLAPACK_t* solverData);
```

5.6.1.2 Solver function

The solver function is the implementation of the `solveNormalEquations()`. For this solver, it is very similar to the MATLAB implementation shown in Listing 4.4. The full implementation of the solver function for the standard form is shown on Listing 5.13.

When called, the function checks if the factorization has been computed for the current interior point iteration, k. If the factorization has not been computed for the current iteration, it computes the normal equation matrix and uses the LAPACK factorization function, `dpotrf()`, to compute the factorization and stores it in the solver data structure. The dispatch function, `ddimm()`, is a custom function in the dispatcher, which computes the matrix-matrix multiplication of a diagonal matrix, stored as a vector, with a general dense matrix. This function is essentially the diagonal-matrix times dense matrix described in Section 4.2.3.5, but it has been generalized to compute either AD or DA, where D is a diagonal matrix and A is a general dense matrix..

Once the factorization has been computed, or if already computed in the current iteration, it uses the LAPACK triangular solve function, `dpotrs()`, to solve the linear system. Since the factorization is saved in the solver data, subsequent calls in the same iteration (same k) does not need to recompute the factorization.

All the operations are done through the dispatcher, which allows the solver to be used for both the CPU and GPU.

5.6.2 CHOLMOD

The CHOLMOD-based linear solver implementation uses CHOLMOD [Dav], which we introduced in Section 5.5.2, to compute the Newton direction by solving the normal equations system.

As mentioned in, this solver is only for the CPU, but the latest version can use NVIDIA GPUs for dense sub-blocks in the sparse matrix. However, we have not used this feature in our tests as we got segmentation fault errors when linking with the GPU-enabled CHOLMOD library.

Listing 5.13: Solver function for LAPACK-based solver

```
void ipmSolverLapackSolveNormalEquationsStdform(const int k,
    ↪ const ipmForm_t form, const double* D, const double E,
    ↪ const double* b, double* x, const void* data) {
  ipmSolverLAPACK_t* solverData = (ipmSolverLAPACK_t*) data;
  dispatch_t* vtable = solverData->vtable;
  int m = solverData->m;
  int n = solverData->n;

  if (solverData->iteration != k) {
    vtable->dcopy(m*n, solverData->A, 1, solverData->AD, 1);
    vtable->ddimm('R', m, n, 1.0, D, solverData->AD, m);
    vtable->dgemm('N', 'T', m, m, n, 1.0, solverData->AD, m,
        ↪ solverData->A, m, 0.0, solverData->L, m);
    if (E != 0.0)
      vtable->daddx(m, E, solverData->L, m+1);
    vtable->dpotrf('L', m, solverData->L, m);
    solverData->iteration = k;
  }
  vtable->dcopy(m, b, 1, x, 1);
  vtable->dpotrs('L', m, 1, solverData->L, m, x, m);
}
```

5.6.2.1 Solver data

The solver data for the CHOLMOD-based solver contains the `cholmod_common` data structure, which is the workspace for CHOLMOD initialized by calling `cholmod_start()`, as well as a pointer to the CHOLMOD loaded constraints matrix, a pointer to the computed factorization, and a counter marking the iteration the factorization was computed in.

The data structure can be initialized with one of the two create functions shown in Listing 5.14. The only difference between the two create functions is how the constraints matrix is loaded. The Matrix Market create function accepts an open `FILE` pointer to a file containing the constraints matrix in Matrix Market format [NIS]. The triplet function takes the constraints matrix as a sparse matrix stored in COO format, where the `val` parameter is an array of all the non-zero elements in the constraints matrix, and `rowInd` and `colInd` are the corresponding row and column indices of each element, respectively.

The create function initializes the CHOLMOD common structure, and loads the constraints matrix into a CHOLMOD sparse matrix. Once the constraints matrix is loaded, it computes the one-time analysis of the matrix required prior to factorization of the normal equations matrix using `cholmod_analyze()`. For an inequality-constrained optimization problem, the constraints matrix is stored transposed, as this makes it possible to use the same solver function as the standard form. Like all the other solvers, the allocated memory in solver data can be released with the corresponding destroy function.

Listing 5.14: Solver data functions for CHOLMOD-based solver

```
void ipmSolverCholmodCreateFromMatrixMarketFile(
    ↪ ipmSolverCholmodData_t* data, const ipmForm_t form,
    ↪ FILE* file);
void ipmSolverCholmodCreateFromTriplet(
    ↪ ipmSolverCholmodData_t* data, const ipmForm_t form,
    ↪ const int m, const int n, const int nnz, const int*
    ↪ rowInd, const int* colInd, const double* val);
void ipmSolverCHOLMODDestroy(ipmSolverCHOLMODData_t* data);
```

5.6.2.2 Solver function

The `ipmSolverCHOLMODSolveNormalEquations()` function implements the solution of the normal equations system. Like the LAPACK-based solver, it checks if the factorization has been computed in the current interior point iteration. If it has not been computed, then it computes $G = AD^{1/2}$ by taking the square root of the elements in D and scaling the columns in A with `cholmod_scale()`. Once G has been computed, the factorization of the regularized normal equations matrix

$$GG^T + E = AD^{1/2}D^{1/2}A^T + E = ADA^T + E \tag{5.23}$$

is computed directly by using `cholmod_factorize_p()`. This function can be used to compute the factorization of $\beta I + AA^T$, where we set $\beta = E$ and I is the identity matrix. The factorization function will automatically use the fill-reducing ordering it determined was best during the analysis when the solver data was created.

Once the factorization has been computed, the normal equations system is solved using `cholmod_solve()`. The factorization is saved in the solver data and reused for subsequent solves in the same interior point iteration.

For more detailed information about the functions in CHOLMOD and its operation, we refer to [Dav13].

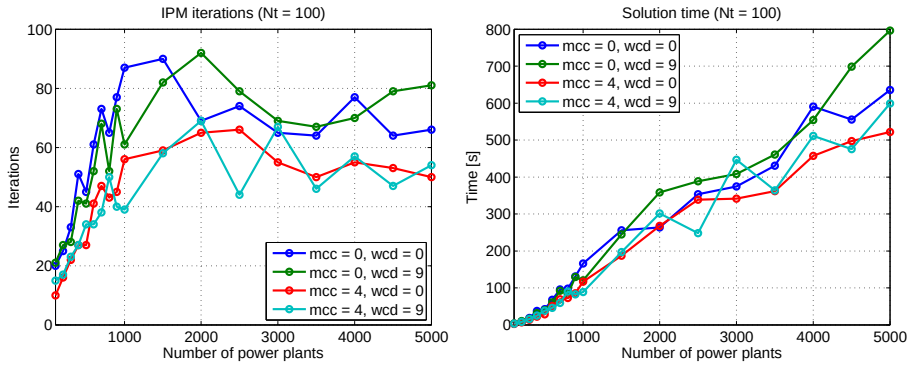
5.7 Usage example

An example of the code required to solve an optimization problem in the standard form, stored on disk, using GPUOPT with the CHOLMOD-based solver is shown in Listing 5.15. The code has been truncated for brevity, such that variable declarations, memory allocation and deallocation, and error checking are not shown. Full examples are available with the GPUOPT source code released along with this thesis.

First a dispatcher is initialized and the problem is loaded from disk. The loaded problem is used to create the solver data for the CHOLMOD-based solver. The interior point context is then created, and the CHOLMOD-based solver is normal equations form solver is assigned to it. Then the problem is solved using the `ipmSolve()` functions and the resulting primal and dual solution is written to disk.

Listing 5.15: Example of using GPUOPT with CHOLMOD-based solver

```
// Create dispatcher
dispatch_t vtable;
dispatchInitialize(&vtable, DISPATCH_VARIANT_HOST);
// Read problem from disk and create CHOLMOD-based solver
ipmSolverCholmodData_t cholmodData;
ioReadMatrixDense(filenameC, &c, NULL, NULL);
ioReadMatrixDense(filenameD, &b, NULL, NULL);
ioReadMatrixSparse(filenameA, &rowIndices, &colIndices,
    ↪ &values, &m, &n, &nnz);
ipmSolverCholmodCreateFromTriplet(&cholmodData,
    ↪ IPM_FORM_STANDARD, m, n, nnz, rowIndices, colIndices,
    ↪ values);
// Create IPM
ipmContext_t ipmContext;
ipmCreateContext(&ipmContext, &vtable, IPM_FORM_STANDARD, m,
    ↪ n, NULL, NULL, c, ipmMatrixCholmodMultiplyAx,
    ↪ ipmMatrixCholmodMultiplyAtx, (void*) &cholmodData);
ipmSetSolver(&ipmContext,
    ↪ ipmSolverCholmodSolveNormalEquations, NULL, (void*)
    ↪ &cholmodData);
// Solve with IPM
ipmSolve(&context, b, x, y);
// Write solutiion to disk
ioWriteMatrixDense(filenameX, x, n, 1);
ioWriteMatrixDense(filenameY, y, m, 1);
```



(a) Iterations **(b)** Solution time

Figure 5.6: Performance of interior point method for the test problem with the CHOLMOD-based solver with and without multiple centrality correctors and weighted corrector directions. The notation $mcc = 4$ indicates maximum 4 MCC iterations, $wcd = 9$ indicates 9 steps on the line search for weighted corrector directions and 0 means it is disabled.

5.8 Computational results

In this section, we demonstrate the performance of GPUOPT using the CHOLMOD-based solver for our test problem from Chapter 3 in the standard form. As the CHOLMOD-based solver is only for the CPU, the results are only for the version of the solver. The results on the GPU are demonstrated in Chapter 6 with the conjugate gradient solver.

As termination tolerances, we have used $tol_p = 10^{-8}$, $tol_d = 10^{-8}$ and $tol_o = 10^{-8}$. The number of iterations and time required to converge to the termination tolerances for the interior point method are shown on Figure 5.6a and Figure 5.6b, respectively.

Multiple centrality correctors reduces the number of iterations required to converge, which translates directly into reduced solution time when using a direct solver. The MCC iterations are very cheap when the factorization is already available, as they can simply reuse the factorization for additional backsolves.

Weighted corrector directions has mixed success with our test problem, sometimes increasing and sometimes reducing the number of iterations required to converge.

5.9 Conclusion

We implemented a modular toolbox for primal-dual interior point algorithm for linear optimization problems with both CPU and GPU support. The implementation was split into three components, the core interior point algorithm, the matrix operations and the linear solving. This modularity made it possible to implement different versions of the matrix operations and linear solving.

The matrix operations were implemented as both dense operations by using BLAS, as in Chapter 4, as sparse operations using CHOLMOD, and as sparse operations in multiple different formats using C. The performance of the different formats were compared on both the CPU and GPU and showed that substantial speed-up can be achieved by utilizing the GPU for sparse matrix-vector operations for our test problem, but that choosing the correct sparse format for a problem is critical for performance.

A linear solver component for the interior point method was implemented using CHOLMOD, which allowed us to compute the Newton directions with a supernodal sparse Cholesky factorization. In the standard form, this substantially increased sparsity of the test problem, and the solution time of the problem was reduced substantially, even compared to the best problem-specific dense GPU solver in Chapter 4. The sparsity allowed us to solve much larger problems, too.

The Mehrotra-predictor corrector interior point method was extended with multiple centrality correctors and weighted corrector directions, and we demonstrated that they can substantially reduce the number of iterations in the interior point method for our test problem. Since multiple centrality correctors can reuse the factorization already computed in the same iteration, this resulted in direct savings in the solution time of the problem for the factorization based solver.

5.10 Future perspectives

The current implementation of GPUOPT only supports linear optimization problems. One of the future improvements we would like to add is support for quadratic optimization problems, which are often seen in model predictive control problems. While the core interior point method has already been updated to support quadratic optimization problems, the solver interface and solvers does not suppport these problems, yet.

GPUOPT currently only contains a sparse Cholesky factorization solver for the CPU by using CHOLMOD. A future improvement would be the inclusion of a GPU-accelerated Cholesky factorization solver. Sparse Cholesky factorization on the GPU has been looked at by multiple authors [GSG+11, ZD12, ZDG+13] and can be accelerated to some extent, however no public libraries are currently available. CHOLMOD also includes support for accelerating parts of the Cholesky factorization with an NVIDIA GPU, but linking with the GPU-enabled version of CHOLMOD resulted in segmentation faults. A new beta version of CHOLMOD has just recently been released, which may address this issue.

Chapter 6

Matrix-Free Preconditioned Conjugate Gradient for GPUOPT

In this chapter, we introduce our implementation of the matrix-free preconditioned conjugate gradient (PCG) solver [Gon12b] as an extension for GPUOPT introduced in Chapter 5.

The PCG solver is an iterative solver which replaces the direct solvers based on Cholesky factorization to compute the Newton directions in the interior point method. The solver is matrix-free, as it it only uses function pointers to operate with the constraints matrix and normal equations matrix, instead of storing the matrices explicitly. This makes it possible to reduce memory usage and solve large sparse and dense systems. The PCG algorithm is also particularly suitable for GPUs, as the main workload is matrix-vector multiplication.

The implementation is validated with our test problem from Chapter 3 and compared to the CHOLMOD-based solver from Chapter 5. We demonstrate that iterative solvers can be competitive with direct solvers in the context of interior point methods, and that the GPU can be used to accelerate the iterative method. We also demonstrate that the regularization and preconditioner introduced by Gondzio in [Gon12b] is key to achieving acceptable performance and convergence

with the iterative solver.

The implementation in this chapter is an extention to our newly developed optimization toolbox, GPUOPT, which we introduced in Chapter 5. One of the key features of this toolbox is that it supports running all the operations on the GPU. However, we only demonstrated the toolbox running on the CPU with the CHOLMOD-based solver as we have not yet presented a linear solver for the GPU, other than the simple LAPACK solver, which cannot solve large problems due to memory requirements. The PCG solver in this chapter makes it possible to run all operations of the interior point method entirely on the GPU, and its performance depends mostly on the matrix-vector multiplication, which can be implemented to exploit any structure present in the constraints matrix.

6.1 Conjugate gradient component

The conjugate gradient (CG) component in our toolbox is an implementation of the preconditioned conjugate gradient algorithm. This component is used by the preconditioned conjugate gradient solver for the linear system of equations, which we describe in Section 6.2.

Conjugate gradient is an iterative method for solving large systems of linear equations in the form

$$Ax = b \tag{6.1}$$

where A is a symmetric, positive-definite matrix. This means we can use conjugate gradient to solve the normal equations form in (5.9). It only requires a method to compute the matrix-vector product with the matrix A and therefore does not require the matrix A to be stored explicitly.

6.1.1 Algorithm

The conjugate gradient algorithm is easy to implement, and is widely described in the literature, so we will only briefly introduce the algorithm in the following section and focus on our particular implementation of it. We refer the reader to [She94] for an introduction to conjugate gradient. Listing 6.1 shows the algorithm for preconditioned conjugate gradient. Conjugate gradient without a preconditioner is equivalent to setting the preconditioner, M, equal to the identity matrix resulting in $z = r$.

Listing 6.1: Preconditioned conjugate gradient algorithm

$r_0 = b - Ax, z_0 = M^{-1}r_0, p_0 = z_0, k = 0$
loop
  $\alpha = \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha p_k$
  $r_{k+1} = r_k - \alpha A p_k$
  if $\left(\frac{r_{k+1}^T r_{k+1}}{r_0^T r_0} < \mathrm{tol}_{\mathrm{cg}}\right)$, **exit loop**
  $z_{k+1} = M^{-1} r_{k+1}$
  $\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$
  $k = k + 1$
end

6.1.2 Implementation

The implementation of the CG component follows the design of the interior point method. A structure called `cgContext_t` is used to hold all the relevant data for a particular linear system and is initialized by calling `cgCreateContext()`. Listing 6.2 shows the the prototype of this function.

Listing 6.2: Creation of conjugate gradient context

```
void cgCreateContext(cgContext_t* context, dispatch_t*
    ↪ vtable, int m, void (*multiplyAx)(const int m, const
    ↪ double* x, double* result, const void* userData),
    ↪ const void* multiplyAxUserData);
```

Like the interior point implementation, it takes a pointer to a dispatch vtable that is used to control whether the implementation runs on the CPU or the GPU. It is also matrix-free, such that it expects a function pointer to a user-defined function, which computes the matrix-vector product, Ap, in the algorithm, and is called once per conjugate gradient iteration. When a context is created, it is initialized with a default tolerance and a maximum number of iterations. They can be specified by the user by setting the associated values in the context. Listing 6.3 on the following page shows the prototype of the function `cgSolve()`, which can be called to solve the linear system of equations.

Listing 6.3: Solving with conjugate gradient component

```
cgStatus_t cgSolve(cgContext_t* context, const double* b,
    ↪ double* x);
```

A right-hand side is passed to the function along with the context defining the problem. The implementation will use the defined dispatcher and matrix-vector function to solve the linear system $Ax = b$, and return the solution in the passed vector x. Various information about the solving is stored in the context, such as achieved accuracy, the number of iterations used, and profiling information. The profiling information shows how much time is spent on the various operations, such as applying preconditioner and computing the matrix-vector multiplication. The accuracy is measured as

$$acc_{\text{cg}} = \frac{r_{k+1}^T r_{k+1}}{r_0^T r_0} \tag{6.2}$$

where r_k is the residual vector for iteration k. The accuracy is used as termination criteria as shown in the algorithm in Listing 6.1.

Calling the solve function after creating the context will result in non-preconditioned conjugate gradient, as no precondtioner has been defined. A precondtioner can be defined by calling the function `cgSetPreconditioner()` before calling solve. Listing 6.4 shows the prototype of this function.

Listing 6.4: Assigning a preconditioner to the conjugate gradient context

```
void cgSetPreconditioner(cgContext_t* context, void
    ↪ (*applyPreconditioner)(const int m, const double* z,
    ↪ double* result, const void* userData), const void*
    ↪ applyPreconditionerUserData);
```

Like the matrix-vector multiplication, the preconditioning is also done with the use of a function pointer to a user-defined function. This function must compute the operation $z = M^{-1}r$. Once assigned, the solver implementation will call this function to apply the preconditioner to a vector. This leaves the memory handling and implementation of the preconditioner to the user.

The full algorithm is quite easy to implement for both CPU and GPU by using the dispatcher, as it can be implemented with BLAS operations. Listing 6.5 shows the implementation of algorithm in `cgSolve()`, though without the error checking and termination check for the sake of brevity. The functions `cgMultiplyAx()` and `cgApplyPreconditioner()` call the user-defined functions defined in the context. If no preconditioner is defined, we simply set $z = r$ which results in the non-preconditioned conjugate gradient.

Listing 6.5: Implementation of conjugate gradient (shortened)

```
if (context->applyPreconditioner == NULL) z = r;
cgMultiplyAx(context, x, r);
vtable->dscal(m, -1.0, r, 1);
vtable->daxpy(m, 1.0, b, 1, r, 1);
double res0 = vtable->ddot(m, r, 1, r, 1);
if (context->applyPreconditioner != NULL)
  cgApplyPreconditioner(context, r, z);
vtable->dcopy(m, z, 1, p, 1);
rz = vtable->ddot(m, r, 1, z, 1);
while (1) {
  cgMultiplyAx(context, p, Ap);
  alpha = rz / vtable->ddot(m, p, 1, Ap, 1);
  vtable->daxpy(m, alpha, p, 1, x, 1);
  vtable->daxpy(m, -alpha, Ap, 1, r, 1);
  context->currentRelativeResidual =
    vtable->ddot(m, r, 1, r, 1) / res0;
  // Termination checks removed for brevity
  if (context->applyPreconditioner != NULL)
    cgApplyPreconditioner(context, r, z);
  rzprev = rz;
  rz = vtable->ddot(m, r, 1, z, 1);
  vtable->dscal(m, rz / rzprev, p, 1);
  vtable->daxpy(m, 1.0, z, 1, p, 1);
}
```

After solving, the context is filled with information about the solution. The user may call the function `cgResetContext()`, which resets the context back to the initialized state, but maintain defined settings such as the preconditioner and tolerances, as well as profiling information. After the context is reset, the solve function can be called with the context again to solve the linear system with a different right-hand side. This is very useful when the solver is needed to solve the same system with different right-hand sides, such as in the interior point algorithm.

6.2 Preconditioned conjugate gradient solver

This linear solver is an implementation of the solver for the matrix-free interior point method presented by Gondzio in [Gon12b], implemented for GPUOPT. In the following sections, we summarize the method and describe the implementation.

6.2.1 Method

As mentioned in Section 5.1.4 on page 74, the ill-conditioning caused by the complementarity matrix in the augmented systems form and normal equations form, due to the partitioning of the complementarity pair when approaching the optimal solution, adversely affects the convergence of iterative methods. In [Gon12b], Gondzio proposes a two-step method for using conjugate gradient to compute the Newton directions.

The first step is to apply primal and dual regularization [AG99] to the optimization problem to bound the condition number of the KKT system. The regularization sets a lower and upper bound of the eigenvalues, such that the values approaching zero are bound to the lower value, and the remaining values are spread between the lower bound and the upper bound.

The second step is to apply a preconditioner to conjugate gradient. The preconditioner is a partial Cholesky factorization, which uses complete pivoting to attempt to eliminate the largest eigenvalues in the matrix [Gon12b]. The Cholesky factorization is truncated after k columns, and only the diagonal of the Schur complement is computed.

A partial Cholesky factorization can be written in the following form

$$G_R = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & I \end{bmatrix} \tag{6.3}$$

where $L = \begin{bmatrix} L_{11} \\ L_{22} \end{bmatrix}$ is the Cholesky factorization of the first k columns of the normal equations matrix, and S is the Schur complement. The preconditioner is computed by using partial Cholesky factorization to eliminate the k largest pivots in the normal equations matrix, where $k << n$. To accomplish this, complete pivoting is used to reorder the normal equations matrix such that

$$d_1 \geq d_2 \geq ... \geq d_k \geq d_{k+1} \geq d_{k+2} \geq ... \geq d_n \tag{6.4}$$

where $D_L = [d_1, ..., d_k]$ and $diag(S) = [d_{k+1}, ..., d_n]$.

The preconditioner is defined as

$$P = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & 0 \\ 0 & D_S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & I \end{bmatrix} \tag{6.5}$$

where $D_S = diag(S)$ is the diagonal of the Schur complement. A key feature of this preconditioner is that it only requires storing and computing k columns and the diagonal.

Notice that if $k = 0$, the preconditioner is equivalent to a diagonal preconditioner with complete pivoting, and if $k = m$, it is equivalent to an exact Cholesky factorization with complete pivoting.

6.2.2 Solver data

Like the other solver implementations in Section 5.6 on page 91, the PCG solver needs to construct a user data structure to hold all the data it requires to solve the linear system. This includes a context for the conjugate gradient component, storage for the preconditioner, as well as additional information required to construct the preconditioner and apply it to a vector. Two helper functions are implemented to initialize and deallocate the user data with the prototypes shown in Listing 6.6.

Listing 6.6: Solver data functions for PCG-based solver

```
void ipmSolverCGCreate(ipmSolverCGUserData_t* data,
    ↪ ipmContext_t* context);
void ipmSolverCGDestroy(ipmSolverCGUserData_t* data);
```

The create function takes a pointer to an interior point context and initializes the passed user data accordingly. The initialized user data is therefore only valid for the particular interior point context, which was passed to the create function. Using the interior point context, the create function is able to create the conjugate gradient context, by using problem information in the IPM context, such as problem dimensions and matrix-vector multiplication functions. The dispatcher used by the IPM context is also used for the conjugate gradient context.

The matrix-vector multiplication functions, `multiplyAx()` and `multiplyAtx()`, in the IPM context are used to construct a matrix-vector multiplication function

for the conjugate gradient context called `ipmSolverCGMultiplyADAtx()`. This function essentially computes the normal equations matrix multiplication with a vector, which is $(ADA^Tx + E)x$ for problems in the standard form.

This is done by first using `multiplyAtx()` to compute A^Tx, then `daxmy()` to compute DA^Tx, and `multiplyAx()` to compute ADA^Tx. The E term is then handled by multiplying it independently with x and adding it to ADA^Tx, resulting in $(ADA^Tx+E)x$. If the IPM context is set to solve a linear optimization problem in the inequality form, a similar function called `ipmSolverCGMultiplyAtDAx()` is constructed to compute $(A^TDAx + E)x$ instead of $(ADA^Tx + E)x$.

The create function only initializes the solver data, such that conjugate gradient can be used to solve the linear system without preconditioner. In order to enable the preconditioner, the function `ipmSolverCGEnablePreconditioner()` must be called. Listing 6.7 shows the prototype for this function.

Listing 6.7: Solver data preconditioner function for PCG-based solver

```
void ipmSolverCGEnablePreconditioner(ipmSolverCGUserData_t*
    ↪ data, int rank, ipmSolverCGPrecondPermute_t permute,
    ↪ void (*getColumnH)(...), void (*getDiagonalH)(...),
    ↪ const void* userData);
```

This function allocates the required memory to store the preconditioner in the solver data. We store the preconditioner of rank k as a $k \times m$ matrix containing L_{11} and L_{21}, and the diagonal containing D_L and D_S is stored as a vector. This limits the storage of the preconditioner to $(m+1) \times k$ double precision numbers. By storing the factorized columns as dense, it is still possible to use standard high-performance BLAS and LAPACK functions to compute the factorization, as well as the triangular solve operations.

Additionally, it also uses the `cgSetPreconditioner()` to define the function `ipmSolverCGApplyPreconditionerCallback()` as the preconditioner function for the conjugate gradient context. This function applies the preconditioner to a vector, which is used to compute $z = M^{-1}r$ in conjugate gradient. Section 6.2.5 describes this in detail.

The `getDiagonalH` and `getColumnH` parameters must be function pointers to user-defined functions, which return the diagonal or a specified column in the normal equations matrix respectively. These functions will be described in further detail in Section 6.2.4, where we describe how the preconditioner is constructed.

6.2.3 Solver function

The solver function, `ipmSolverCGSolveNormalEquations()`, implements the normal equations form solver interface described in Section 5.6. It is similar to the solver functions for the factorization-based solvers, but instead of computing a full factorization, it only computes a preconditioner.

The function checks if the preconditioner has been computed for the current interior point iteration. If the preconditioner has not been computed yet, it computes the preconditioner and stores it in the solver data. Then it calls `cgSolve()` to use the conjugate gradient context in the solver data to compute the Newton direction.

6.2.4 Computing preconditioner

To compute the preconditioner, the solver needs additional information about the normal equations matrix. To determine the permutation of the matrix, it needs access to the diagonal of the normal equations matrix. Furthermore, to construct the preconditioner, it needs access to specific columns in the normal equations matrix. To accomplish this, the solver takes two additional user-defined function pointers with the prototypes shown in Listing 6.8.

Listing 6.8: Interface to PCG-based solver required to construct preconditioner

```
void getDiagonalH(const int m, const int n, const ipmForm_t
    ↪ form, const double* D, double* diagonal, const void*
    ↪ data);
void getColumnH(const int m, const int n, const ipmForm_t
    ↪ form, const int index, const double* D, double*
    ↪ column, const void* data);
```

The `getDiagonalH()` function must compute the diagonal of either ADA^T or A^TDA, depending on the `form` parameter, and store it in the passed `diagonal` parameter as a vector. Similarly, the `getColumnH()` function must compute a column in ADA^T or A^TDA specified by the column index in the `index` parameter.

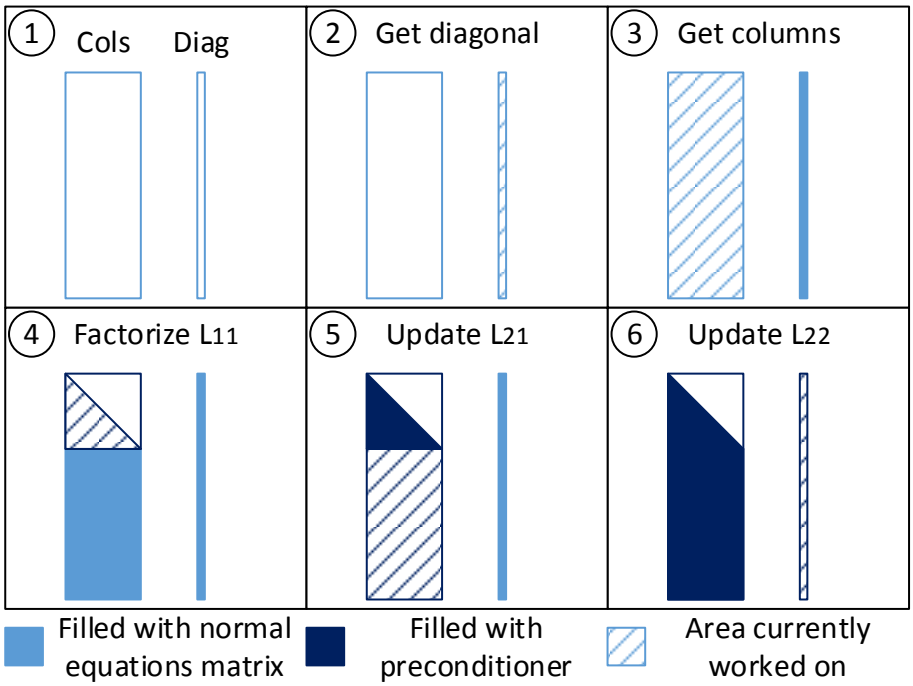


Figure 6.1: Computation of preconditioner for PCG-based solver.

Figure 6.1 shows the steps required to compute the preconditioner in a schematic way and the steps are described below.

1. The first step shows the storage of the first k columns in the preconditioner as a dense $m \times k$ matrix and the diagonal is stored as a vector, which are denoted by `Cols` and `Diag`, respectively.

2. The second step is to compute the diagonal of the normal equations matrix by using the user-defined function `getDiagonalH()` and permuting it according to Equation (6.4). A permutation vector is computed at the same time, which is used to determine the permutation of the normal equations matrix.

3. The third step is to compute the first k columns of the permuted normal equations matrix by using `getColumnH()`. Since the columns are computed in the permuted order, the rows have already been permuted in the preconditioner. The computed columns are then permuted accordingly, such that the columns in the preconditioner are permuted both column-wise and row-wise.

4. Once the required parts of the permuted normal equations matrix are loaded, the fourth step is to factorize the upper $k \times k$ matrix in the columns using the dense Cholesky factorization function `dpotrf()` to compute L_{11} cf. Equation (6.5).

5. The fifth step is to use `dtrsm()` to update the remaining rows to compute L_{21}, as it is done in a normal blocked Cholesky factorization.

6. The sixth and last step is to update the diagonal by computing the diagonal of Schur's complement, D_S, using L_{21}. The computation of the diagonal of Schur's complement cannot be done efficiently using standard BLAS operations as it requires $m - k$ vector dot products, where m is the number of rows in the normal equations matrix. It would be highly inefficient to call BLAS individually for every diagonal element. The CPU implementation is done with a for-loop, while we have implemented a CUDA kernel for the GPU implementation. Listing 6.9 shows the kernel.

Listing 6.9: CUDA kernel to compute diagonal of Schur's complement

```
__global__ void computeSchurApproximationSmallK_kernel(const
    ↪ int M, const int K, const double* cols, double* diag) {
  unsigned int idx;
  for (idx = K + blockIdx.x * blockDim.x + threadIdx.x; idx <
      ↪ M; idx += gridDim.x * blockDim.x) {
    double val = 0.0;
    for (unsigned int i = 0; i < K; i++) {
      val += cols[idx + i*M] * cols[idx + i*M];
    }
    diag[idx] = sqrt(diag[idx] - val);
  }
}
```

Since the rank of the preconditioner, k, is very small, we create a thread for every element in D_S. Every thread computes the dot product for the corresponding row in L_{21}. This can be done without reduction, as only one thread is responsible for a single diagonal element. As the threads iterate over the elements in the rows of L_{21}, they load adjacent values in the columns, resulting in coalesced memory access, as the columns are stored column-wise. Once the dot product has been computed, the diagonal element of the Schur complement is computed by subtracting it from the corresponding diagonal element in the normal equations matrix and computing the square root of the result.

Listing 6.10: Applying preconditioner

```
// Non-transpose solve, L\x
vtable->dtrsv('L', 'N', 'N', k, data->Mcols, m, x, 1);
if (m != k) {
  vtable->dgemv('N', m - k, k, -1.0, data->Mcols + k, m, x,
      ↪ 1, 1.0, x + k, 1);
  vtable->daxdypbz(m - k, 1.0, x + k, 1, data->Mdiag + k, 1,
      ↪ 0.0, x + k, 1);
}
// Transpose solve L'\(L\x)
if (m != k) {
  vtable->daxdypbz(m - k, 1.0, x + k, 1, data->Mdiag + k, 1,
      ↪ 0.0, x + k, 1);
  vtable->dgemv('T', m - k, k, -1.0, data->Mcols + k, m, x +
      ↪ k, 1, 1.0, x, 1);
}
vtable->dtrsv('L', 'T', 'N', k, data->Mcols, m, x, 1);
```

6.2.5 Applying preconditioner

Since the preconditioner is a factorization, it would normally be applied by calling a solve function such as `dpotrs()`. However, because the preconditioner is stored in a custom format, it is necessary to call the operations manually. Listing 6.10 shows the code which implements the solve operation for the preconditioner in our format.

The code first does the non-transposed solve by calling triangular solve for the L_{11} part of the preconditioner, and then solves for L_{21} and D_S by calling matrix-vector multiplication and element-wise vector division respectively. The transposed solve, which is equivalent to the solve with the upper Cholesky factor, is done in the opposite order with the transpose matrix parameter to `dgemv()` and `dtrsv()`.

When applying the preconditioner to a vector, it is necessary to first permute the vector according to the permutation of the preconditioner, then apply the preconditioner, and then reverse the permutation of the resulting vector. This is required, because the conjugate gradient solver is operating with the non-permuted system. This cannot be avoided, as we do not have a matrix-vector multiplication function for the constraints matrix, which can operate with the permuted system.

An alternative approach would be to implement code, that applies the precon-

ditioner, to take a permutation vector and index elements accordingly, without directly permuting the vector. This would not be possible with BLAS operations and would require manual implementation of the triangular solve, matrix-vector multiplication, and element-wise vector division operations for both the CPU and GPU.

As the preconditioner is applied in every iteration of the conjugate gradient method, it is important for performance to have very efficient vector permutation operations.

6.2.6 Handling permutation

The permutation operations in our implementation are done with the library Thrust [NVId]. Thrust is a very efficient library for data manipulation on NVIDIA GPUs, however it also includes support for the same operations on the CPU. It is a high-level C++ interface, which makes it easy to use.

The permutation is computed by using Thrust's key-value sort as shown in Listing 6.11.

Listing 6.11: Using Thrust to compute permutation vector

```
thrust::sort_by_key(x, x + data->m, perm,
    ↪ thrust::greater<double>());
```

The `x` parameter is set to the diagonal, while `perm` parameter is an integer array initialized to hold the indices $1, 2, \ldots, m$. When Thrust sorts the diagonal, the elements in `perm` are permuted in the same order as the elements in x, resulting in an index vector that can be used as a permutation vector.

Once the permutation is computed, it can be applied to a vector by using the `gather()` function as shown in Listing 6.12.

Listing 6.12: Using Thrust to apply permutation vector

```
thrust::gather(perm, perm + data->m, x, y);
```

The inverse permutation can be computed and applied in the same way. The inverse permutation is used to reverse the permutation applied to a vector.

Table 6.1: Matrix-free functions

getColumnH()	Must return a specified columns in ADA^T or A^TDA.
getDiagonalH()	Must return the diagonal of ADA^T or A^TDA and as a vector.
multiplyAx()	Must return the result of the matrix-vector multiplication of the constraints matrix A and a vector x.
multiplyAtx()	Must return the result of the matrix-vector multiplication of the transposed constraints matrix A^T and a vector x.

6.3 Problem-specific implementation of test case

In this section, we describe the implementation of the test case from Chapter 3 as a problem-specific implementation of the two matrix-vector multiplication functions, required by the interior point method, as well as the two normal equations matrix functions, required by the preconditioned conjugate gradient solver. Table 6.1 summarizes the four functions.

6.3.1 Matrix-vector multiplication functions

The inequality form of the test case is described in Section 3.2 on page 28 and the problem-specific matrix-vector multiplication and matrix-transpose-vector multiplication are already described in Section 4.3.1.1 on page 58.

The standard form of the test case (described in Section 3.3 on page 32) is obtained by introducing slack variables to the inequality constraint to express the constraints as equality constraints. This only changes the matrix-vector multiplication slightly, as the added slack variables are simply added to result, such that the matrix-vector multiplication is computed as

$$\hat{A}\begin{bmatrix}\hat{x}_1\\ \hat{x}_2\\ \hat{x}_3\\ \hat{x}_4\\ \hat{x}_5\\ \hat{x}_6\\ \hat{x}_7\\ \hat{x}_8\end{bmatrix} = \begin{bmatrix}b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\end{bmatrix} = \begin{bmatrix}\hat{x}_1 - \hat{x}_3\\ -\hat{x}_1 - \hat{x}_4\\ \hat{x}_2 - \hat{x}_5\\ \Psi\hat{x}_1 - \hat{x}_6\\ -\Psi\hat{x}_1 - \hat{x}_7\\ \Gamma\hat{x}_1 + \hat{x}_2 - \hat{x}_8\end{bmatrix} \tag{6.6}$$

The matrix-transpose-vector multiplication is computed as

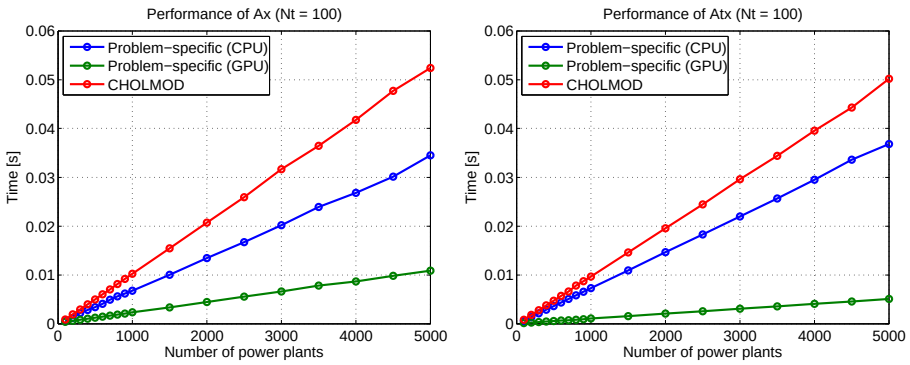
$$\hat{A}^T \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{x}_5 \\ \hat{x}_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix} = \begin{bmatrix} \hat{x}_1 - \hat{x}_2 + \Psi^T(\hat{x}_4 - \hat{x}_5) + \Gamma^T \hat{x}_6 \\ \hat{x}_3 + \hat{x}_6 \\ -\hat{x}_1 \\ -\hat{x}_2 \\ -\hat{x}_3 \\ -\hat{x}_4 \\ -\hat{x}_5 \\ -\hat{x}_6 \end{bmatrix} \tag{6.7}$$

The GPU implementation of the matrix-vector computation is similar to the code shown in Listing 4.16 on page 63, except it has an additional `daxpy()` call to add the new slack variables. The matrix-transpose-vector is similarly implemented for the problem in standard form. We use the `userData` parameter passed to the interior point context during initialization, to pass the Γ matrix to the matrix-vector multiplication functions. We pass Γ as a full dense matrix despite its structure, as this allows us to use a single BLAS call to compute the matrix-vector multiplication. Due to the block Toeplitz structure of Γ, it would also be possible to simply store only the first N_p columns and implement the matrix-vector multiplication functions for the unique structure of Γ. This is not currently done, but should be considered as a future improvement as it reduces both memory usage and required memory bandwidth.

The matrix-vector functions have been implemented on both the CPU and the GPU using CBLAS and CUBLAS respectively, so we can compare the performance of the two implementations.

6.3.1.1 Performance results

We have tested the performance of the problem-specific matrix-vector multiplication for our test case, and compared it on the CPU and GPU with the CHOLMOD-based matrix-vector multiplication functions, used for the tests in Chapter 5. Like the tests in Chapter 4, the test was run on the test machine as described in Section 1.4 on page 4. The performance of the matrix-vector product and the matrix-transpose-vector product is shown in Figure 6.2a and 6.2b, respectively. The CPU code is executed sequentially.



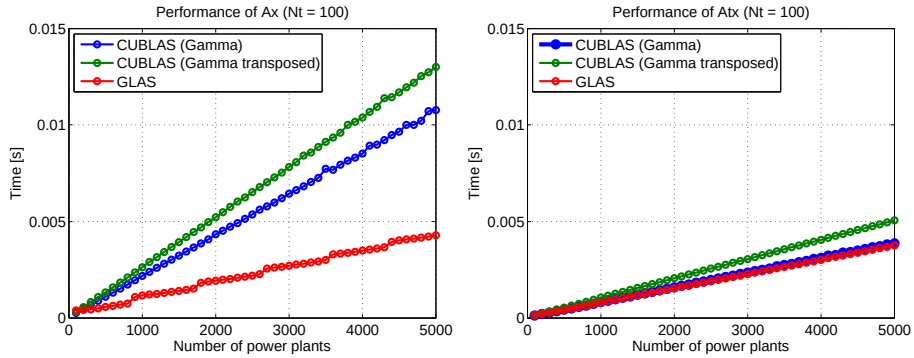
(a) $\hat{A}\hat{x}$ **(b)** $\hat{A}^T\hat{x}$

Figure 6.2: Performance of the problem-specific matrix-vector multiplications.

The problem-specific implementations are clearly faster than doing general sparse matrix-vector multiplication with CHOLMOD. This is expected, as the problem-specific implementation exploits the problem structure. While CHOLMOD and the problem-specific CPU version have similar performance for the matrix-vector and matrix-transpose-vector product, the GPU version shows significantly worse performance for the matrix-vector product compared to its matrix-transpose-vector product. This is due to the fact that the major computational task in the matrix-vector product is the multiplication with the dense Γ submatrix. The dimensions of Γ, as we have previous mentioned, is $(N_t) \times (N_t \times N_p)$, which makes it a very wide matrix for the case with a large number of power plants.

In [Sø12], Sørensen demonstrates that the performance of the matrix-vector product in CUBLAS is optimized for matrices which are mostly square, and that the performance for wide matrices is very low. Additionally, he implements an auto-tuning framework and demonstrates, that by implementing different matrix-vector kernels for matrices of different shapes, it is possible to achieve much better performance for the wide and tall matrices. The auto-tuning framework also determines the optimal number of elements per thread, and number of threads to create. The improved kernels are implemented and are available in GLAS.

The available download of GLAS [Sø] provides an auto-tuned version for a NVIDIA Tesla C2050 card, and only for matrices with up to one million rows or columns. Instead of using the auto-tuned library for a different card with a limitation of the dimensions of the matrices and vectors, we have extracted two well-performing kernels for wide and tall matrices and employed them in our matrix-vector functions. We have only extracted the non-transpose matrix-

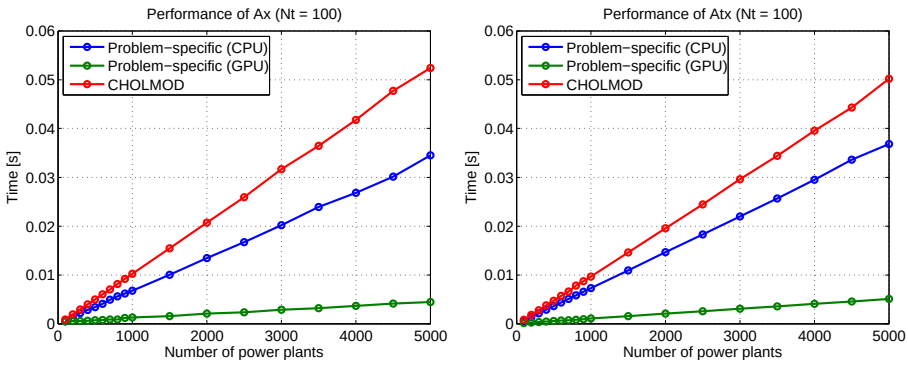


(a) $\hat{A}\hat{x}$ **(b)** $\hat{A}^T\hat{x}$

Figure 6.3: Performance of matrix-vector multiplication with Γ on the GPU with CUBLAS and GLAS.

vector multiplication functions. Thus, to use the GLAS kernels for doing matrix-vector multiplication, it is required to store Γ as a transposed matrix. This doubles the memory requirement, though. Figure 6.3a and Figure 6.3b show the performance of CUBLAS with both the non-transposed stored Γ and the transposed stored Γ. They also show the performance of GLAS. GLAS uses the non-transposed Γ for $\hat{A}\hat{x}$ and the transposed Γ for $\hat{A}^T\hat{x}$.

For the matrix-vector product, $\hat{A}\hat{x}$, GLAS displays much better performance than CUBLAS, regardless of whether the matrix is stored normally or transposed. The performance of GLAS matrix-vector multiplication is actually very similar to the performance of the matrix-transpose-product, $\hat{A}^T\hat{x}$, just like the CPU versions had similar performance for both matrix-vector multiplication and matrix-transpose vector multiplication. For the matrix-transpose-vector product, CUBLAS has similar performance to GLAS. This means we can achieve optimal performance and still avoid storing Γ as transposed. We store only Γ and use GLAS for $\hat{A}\hat{x}$ and CUBLAS for $\hat{A}^T\hat{x}$. Figure 6.4 on the following page shows the performance of the problem-specific matrix-vector multiplication when GLAS is used. Compared to the results in Figure 6.2, the matrix-vector multiplication for the problem-specific implementation on the GPU is now as fast as the matrix-transpose-vector multiplication.

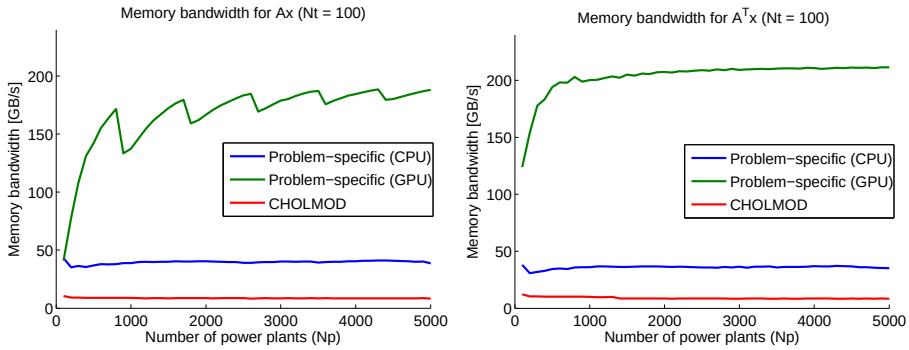


(a) $\hat{A}\hat{x}$ **(b)** $\hat{A}^T\hat{x}$

Figure 6.4: Performance of the problem-specific matrix-vector multiplications with GLAS instead of CUBLAS for the matrix-vector multiplication on the GPU.

The effective memory bandwidth (as defined in Section 5.5.4.2 on page 90) of the problem-specific matrix-vector multiplication functions are shown in Figure 6.5.

The memory bandwidth of the test machine is approximately 42 GB/s for the CPU and 208 GB/s for the GPU, theoretically. Unlike the general sparse formats in Section 5.5.3 on page 87, there is no memory bandwidth wasted on index vectors for the problem specific implementation. It fully maximizes the memory bandwidth of the CPU and GPU for the two operations. The matrix-transpose-vector multiplication on the GPU actually exceeds the GPUs bandwidth slightly, which is due to cache effects.



(a) $\hat{A}\hat{x}$ **(b)** $\hat{A}^T\hat{x}$

Figure 6.5: Memory bandwidth of the problem-specific matrix-vector multiplications with GLAS instead of CUBLAS for the matrix-vector multiplication on the GPU.

6.3.2 Normal equations matrix functions

The normal equations matrix for the test problem in inequality form is described in Section 4.3.1.2 on page 58.

For the standard form of the problem, the columns of the normal-equations matrix $H = \hat{A}D\hat{A}^T$ can be computed in two different ways. The simplest method of computing a column in H is to use the matrix-vector functions, `multiplyAx()` and `multiplyAtx()`. Column k in H can be computed by multiplying H with a vector where the element $x_k = 1$ and the remaining values are zero. This is inefficient, however, as it requires two full matrix-vector multiplications per column.

A more efficient method is to implement the function such that a column is computed according to its index. For our test case in the standard form, the normal equations matrix can be computed in the following way

$$\hat{H} = \hat{A}D\hat{A}^T = \begin{bmatrix} \hat{H}_{11} & \hat{H}_{21}^T & \hat{H}_{31}^T & \hat{H}_{41}^T & \hat{H}_{51}^T & \hat{H}_{61}^T \\ \hat{H}_{21} & \hat{H}_{22} & \hat{H}_{32}^T & \hat{H}_{42}^T & \hat{H}_{52}^T & \hat{H}_{62}^T \\ \hat{H}_{31} & \hat{H}_{32} & \hat{H}_{33} & \hat{H}_{43}^T & \hat{H}_{53}^T & \hat{H}_{63}^T \\ \hat{H}_{41} & \hat{H}_{42} & \hat{H}_{43} & \hat{H}_{44} & \hat{H}_{54}^T & \hat{H}_{64}^T \\ \hat{H}_{51} & \hat{H}_{52} & \hat{H}_{53} & \hat{H}_{54} & \hat{H}_{55} & \hat{H}_{65}^T \\ \hat{H}_{61} & \hat{H}_{62} & \hat{H}_{63} & \hat{H}_{64} & \hat{H}_{65} & \hat{H}_{66} \end{bmatrix} \tag{6.8}$$

where

$$\hat{H} = \begin{bmatrix} D_1 + D_3 & \hat{H}_{21}^T & \hat{H}_{31}^T & & \\ -D_1 & D_1 + D_4 & \hat{H}_{32}^T & & \\ 0 & 0 & D_2 + D_5 & & \\ \Psi D_1 & -\Psi D_1 & 0 & \cdots & \\ -\Psi D_1 & \Psi D_1 & 0 & & \\ \Gamma D_1 & -\Gamma D_1 & D_2 & & \end{bmatrix} \tag{6.9}$$

$$\begin{bmatrix} \hat{H}_{41}^T & \hat{H}_{51}^T & \hat{H}_{61}^T \\ \hat{H}_{42}^T & \hat{H}_{52}^T & \hat{H}_{62}^T \\ \hat{H}_{43}^T & \hat{H}_{53}^T & \hat{H}_{63}^T \\ \cdots \quad \Psi D_1 \Psi^T + D_6 & \hat{H}_{54}^T & \hat{H}_{64}^T \\ -\Psi D_1 \Psi^T & \Psi D_1 \Psi^T + D_7 & \hat{H}_{65}^T \\ \Gamma D_1 \Psi^T & -\Gamma D_1 \Psi^T & \Gamma D_1 \Gamma^T + D_2 + D_8 \end{bmatrix}$$

The diagonal of the normal equations matrix is the diagonal of the matrices H_{ii} for $i = 1, \ldots, 6$. It can be computed as

$$\operatorname{diag}(\hat{A} D \hat{A}^T) = \begin{bmatrix} d_1 + d_3 \\ d_1 + d_4 \\ d_2 + d_5 \\ d_1 + \begin{bmatrix} 0 & d_1(N_p : end) \end{bmatrix} + d_6 \\ d_1 + \begin{bmatrix} 0 & d_1(N_p : end) \end{bmatrix} + d_7 \\ \operatorname{diag}(\Gamma D_1 \Gamma^T) + d_2 + d_8 \end{bmatrix} \tag{6.10}$$

where N_p is the number of power-plants in the system, $D_i = \operatorname{diag}(d_i)$ and $\operatorname{diag}(D) = [d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8]$ corresponding to the columns in $\hat{A}$.

6.4 Computational results

We have run GPUOPT with the test case described in Chapter 3 using the test machine described in Section 1.4.

6.4.1 Standard form

In this section, we describe the results when solving our test case problem in standard form.

6.4.1.1 Convergence

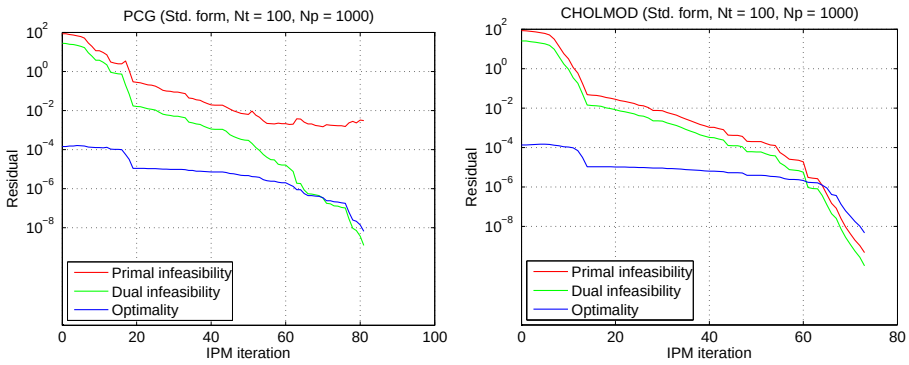
To demonstrate the convergence of the solver, we show the residuals for a specific problem size with a prediction horizon of 100 time steps and 1000 power plants. Figure 6.6 shows the three residuals in each interior point iteration for both the PCG solver and the CHOLMOD-based solver.

The CHOLMOD-based solver manages to bring all the residuals below 10^{-8}, which was the termination criteria in Section 5.8. Unfortunately, the primal infeasibility stalls a bit below 10^{-2} for the PCG solver. This is due to the inaccuracy of the PCG solver.

For a problem in standard form, the PCG solver solves the normal equations form to compute the dual direction, Δy, as shown in (5.9). The PCG solver only solves this to an accuracy of 10^{-5}, and then computes the primal and optimality direction from this result. While the dual infeasibility is reduced as the interior point method progresses, the inaccuracy in the search direction seems to stall the primal infeasibility around $\sqrt{10^{-5}}$. Due to this, we set the termination tolerances to $\text{tol}_\text{p} = 10^{-2}$, $\text{tol}_\text{d} = 10^{-2}$ and $\text{tol}_\text{o} = 10^{-8}$ when using the PCG solver. We accept that we can not solve the system as accurately as the factorization-based solver.

6.4.1.2 Regularization and preconditioning

The regularization and preconditioning are critical for the performance of the PCG solver. Figure 6.7 illustrates this by showing the convergence of the interior point method for our test problem with 100 time steps and 1000 power



(a) PCG (Standard form) **(b)** CHOLMOD (Standard form)

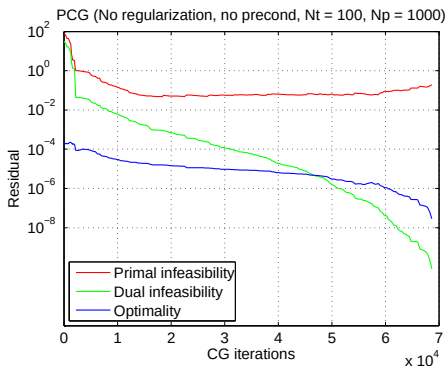
Figure 6.6: Convergence of the PCG solver compared to CHOLMOD solver for the test case problem in the standard form. Multiple centrality correctors and weighted corrector directions are not used.

plants, when using the PCG solver with and without regularization and preconditioning. The x-axis shows the total number of conjugate gradient iterations, as the computational cost is dependent on the number of CG iterations, unlike a factorization-based solver, where the performance is dependent on the number of factorizations, and thus the number of IPM iterations.

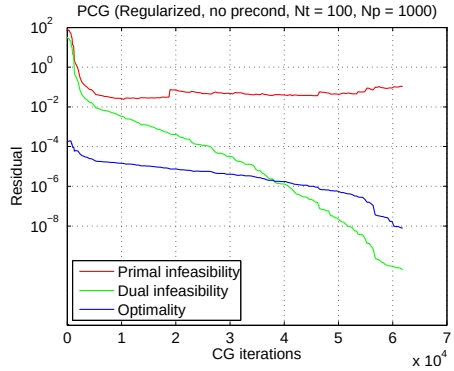
Figure 6.7a: Convergence without regularization and preconditioning. The method never converged to the tolerances defined in the previous section. It used 62394 CG iterations before the optimality reached the tolerance 10^{-8}, and the interior point method terminated after 68663 CG iterations, without reducing the primal and dual infeasibilities below the defined tolerances.

Figure 6.7b: Convergence with regularization and without preconditioning. It performed only slightly better by using 60283 CG iterations before optimality reached its tolerance, but the interior point method terminated after 61869 CG iterations, without reducing the primal and dual infeasibilities below the defined tolerances.

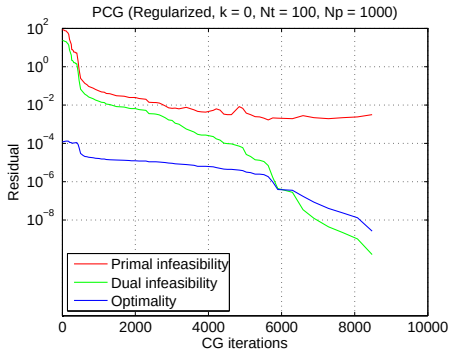
Figure 6.7c: Convergence with regularization and diagonal preconditioning. The interior point method converged in only 8481 CG iterations. This clearly shows that even a simple diagonal preconditioner works well to precondition conjugate gradient, when used in an interior point method.



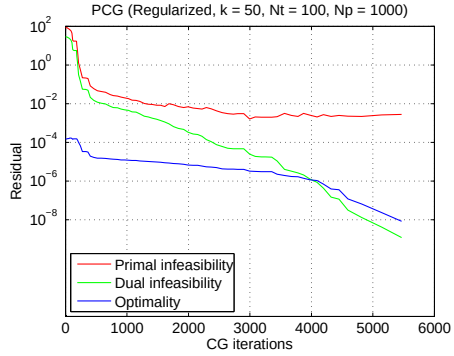
(a) No regularization and preconditioning



(b) Regularized and no preconditioning



(c) Regularized and preconditioned, $k = 0$



(d) Regularized and preconditioned, $k = 50$

Figure 6.7: Convergence of the PCG solver with/without regularization/preconditioning for the test problem with a prediction horizon of 100 time steps and 1000 power plants.

Figure 6.7d: Convergence with regularization and preconditioning of rank 50. This reduces the number of CG iterations required for the interior point method to converge to only 5467. By increasing the rank of the preconditioner, the number of CG iterations can be further reduced at the cost of computing, storing and applying a preconditioner in every CG iteration.

6.4.1.3 Multiple centrality correctors and weighted corrector directions

One of the main reasons to apply multiple centrality correctors [Gon96] is the reuse of the expensive factorization in every interior point iteration when using a direct solver. This makes it possible to reduce the number of total factorizations required, and thus reduces the cost of solving the optimization problem.

When using an iterative solver, there is no factorization to reuse, and computing a centrality corrector can be just as expensive as any other search direction. Figure 6.8 shows the number of IPM iterations required to converge, and how many CG iterations were used to reach each IPM iteration, for the same problem with and without multiple centrality correctors and weighted corrector directions.

Enabling multiple centrality correctors and weighted corrector directions with the PCG solver produces the same results as enabling them with the CHOLMOD-based solver, as described in Section 5.8. Using both multiple centrality correctors and weighted corrector directions results in the smallest number of IPM iterations. It also uses the most CG iterations to compute, however, and the solution time when solving with the PCG solver is dependent on the number

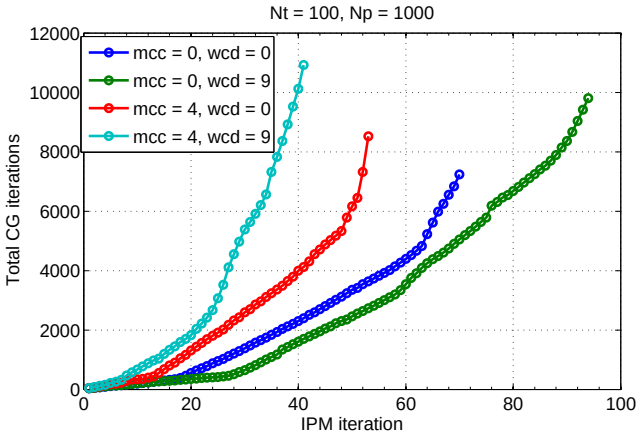


Figure 6.8: Performance of the PCG-based solver with and without multiple centrality correctors and weighted corrector directions, for a test problem with prediction horizon of 100 time steps and 1000 power plants. The plot shows the total number of CG iterations compared to the interior point method. The legend denotes how many MCC iterations and how many steps in the weighted corrector directions are allowed. All four tests were run until they converged.

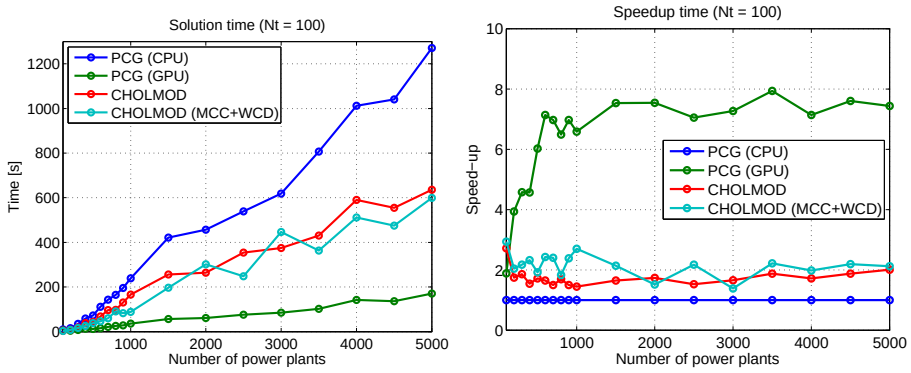
of CG iterations, not the number of IPM iterations. The lowest solution time is achieved, when both multiple centrality correctors and weighted corrector directions are disabled, despite the additional IPM iterations.

In all our subsequent tests with the PCG solver, we do not use multiple centrality correctors and weighted corrector directions. Gondzio allowed the interior point method to use one or two centrality correctors to stabilize the interior point method [Gon12b], but this is not necessary for our test problem.

6.4.1.4 Performance

We solve the test problem with a prediction horizon of 100 time steps and with a varying number of power plants from 100 to 5000 power plants. The termination tolerances used are $\text{tol}_\text{p} = 10^{-2}$, $\text{tol}_\text{d} = 10^{-2}$ and $\text{tol}_\text{o} = 10^{-8}$. Both primal and dual regulalization is set to 10^{-8} and a preconditioner of rank $k = 10$ is used. The PCG solver is set to a maximum of 200 iterations with a termination tolerance $\text{tol}_\text{cg} = 10^{-5}$.

Figure 6.9 shows the performance of the interior point method with the PCG solver. The PCG solver using the CPU is run sequentially, as GPUOPT has not yet been properly optimized for multi-core CPUs.



(a) Solution time **(b)** Speed-up

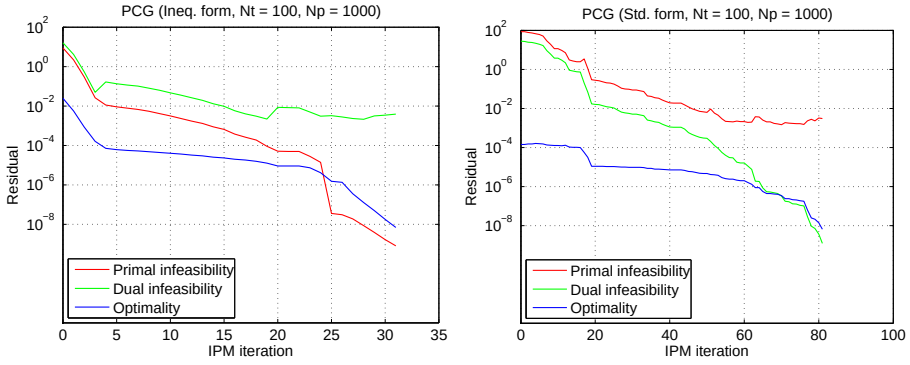
Figure 6.9: Performance of the PCG-based solver compared to CHOLMOD-based solver with our test case problem in the standard form. The PCG solver using a CPU is used as a reference for the speed-up plot.

The CHOLMOD-based solver is approximately two times faster than the sequential CPU implementation of the PCG solver. The factorization-based solver can utilize efficient sparsity preserving orderings, due to the high sparsity of the normal equations matrix. For our test problem, it uses minimum degree (AMD) ordering. This results in a fast factorization, which the PCG solver cannot compete against. Gondzio also remarks, that he does not expect the PCG solver to be competitive for sparse problems [Gon12b].

While the sequential CPU version of the PCG solver is slower than the CHOLMOD-based solver, the benefit of the PCG solver is that it is much easier to parallelize using a GPU. The GPU implementation of the PCG solver shows a 7.5× speed-up compared to the sequential CPU version and about 3.5× speed-up compared to the CHOLMOD-based solver, despite the sparsity of the problem.

6.4.2 Inequality form

The test problem in standard form has five times as many decision variables as the problem in the inequality form, due to the introduced slack variables. In Chapter 5, we switched to the standard form because of the need to maintain sparsity in our normal equations matrix and Cholesky factor, when solving large problems with a direct method. However, since the conjugate gradient solver does not explicitly form the normal equations matrix, it can solve the problem in inequality form instead. In this section, we compare the performance of solving



(a) PCG (Inequality form) **(b)** PCG (Standard form)

Figure 6.10: Convergence of the PCG solver compared to CHOLMOD solver for the test case problem in the standard form. Multiple centrality correctors and weighted corrector directions not used.

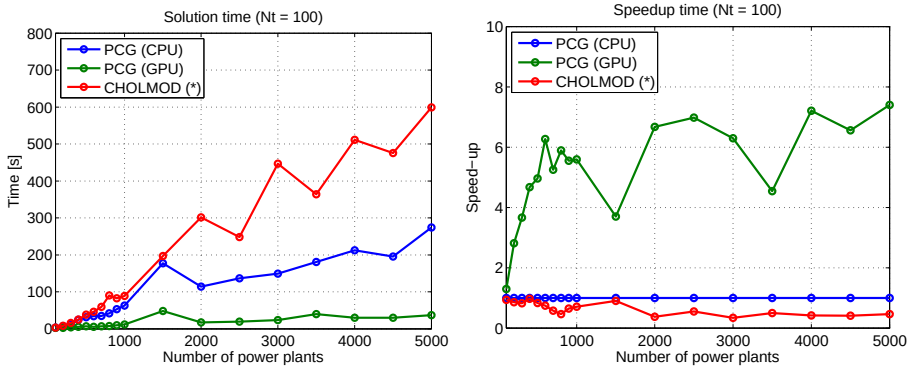
the problem in inequality form with the performance of solving the problem in the standard form.

6.4.2.1 Convergence

Like the convergence for the problem in standard form, described in Section 6.4.1.1, the infeasibility in the PCG solver also stalls when solving the problem in inequality form. This is shown in Figure 6.10b. Instead of the primal infeasibility stalling, it is the dual infeasibility which stalls, as the solver for the inequality form solves the normal equations system for the primal search direction. This is shown on Figure 6.10a

6.4.2.2 Performance

We have run the same test as in Section 6.4.1.4, but with the test problem in inequality form. The performance is shown on Figure 6.11. We compare the solution time of the PCG solver for the inequality form with the CHOLMOD-based solver for the the standard form, as it is not possible to solve the problem in the inequality form with the CHOLMOD-based solver. With a prediction horizon of 100 time steps and 5000 power plants, the normal equations matrix has a dense window with the dimensions 500000×500000, which would require around 1863 GB to store in memory.



(a) Solution time **(b)** Speed-up

Figure 6.11: Performance of the PCG-based solver when solving the problem in the inequality form (3.10). The CHOLMOD-based solver it is compared with solves the standard form problem (3.28).

Due to the smaller problem size in the inequality form, the sequential CPU version of the PCG solver finds the solution about 2× faster than the CHOLMOD-based solver for larger problems. The GPU implementation shows a speed-up between 5× to 7.5×, depending on the problem size.

6.5 Conclusion

In this chapter, we described the implementation of a conjugate gradient component for GPUOPT, and used the component to implement a preconditioned conjugate gradient solver to compute the Newton directions in the interior point method.

The preconditioned conjugate gradient solver is based on [Gon12b], and utilized a generic preconditioner for solving the normal equations system in an interior point method. The solver was implemented to only use function pointers to handle the constraints matrix and normal equations matrix. This allows the solver to be applied to very large problems, even when the normal equations matrix is dense, as it never explicitly forms the matrix.

We demonstrated the performance of the iterative solver using both the CPU and GPU, and compared it to the CHOLMOD-based direct solver described in Section 5.6.2. For the test case problem in standard form, which is sparse, the sequential CPU implementation of the PCG solver fails to solve faster than the CHOLMOD-based solver. However, the GPU implementation shows a good speed-up of up to 8×, compared to the sequential CPU version, and on average a 3.5× speed-up compared to the CHOLMOD-based solver.

For the test case problem in the inequality form, the CHOLMOD-based solver can not solve it at all, due to the memory required to store the dense normal equations matrix. This is not a problem for the PCG solver, and the sequential CPU implementation solved the optimization problem 2× faster than the CHOLMOD-based solver can solve the equivalent problem in standard form. The GPU implementation showed a speed-up of 4× to 7.5× compared to the sequential CPU implementation, and an impressive 9× to 18× speed-up compared CHOLMOD-based solver.

Our results showed that the PCG solver can be accelerated efficiently with a GPU. While we only compared with a sequential CPU version, the problem-specific CPU implementation of both the matrix-vector multiplication and matrix-transpose-vector multiplication already fully utilizes the memory bandwidth of the CPU. Since the operations are memory-bound, using multiple cores should

not give a large difference in performance. Like in Chapter 4, the best speed-up on the GPU is achieved when the problem is dense. By using an iterative method, it is possible to solve these dense problems with very limited memory.

6.6 Future perspective

The preconditioning of the solver is currently implemented as a dense factorization and columns are stored as dense. This does not present a problem, as the rank of the preconditioner is generally very small and easily fits in memory. As a future improvement, the preconditioner could be implemented as sparse which could be beneficial if the normal equations matrix is very sparse. This would reduce memory usage, and could also speed-up the preconditioner operations.

Another important future improvement is the completion of the multi-core implementation of the solver. While this thesis focuses on using the GPU, for some optimization problems the CPU may be better suited, and a combination of using both the CPU and GPU should be explored for maximum performance.

Chapter 7

Conclusion

Over the recent years, development in hardware, and especially software, has made it feasible to use graphics cards (GPUs) as a computing resource for certain types of calculations. This resource comes (almost) for free on computers, that are equipped with a graphics card as a standard component, such as desktop or laptop computers, but there are also special high-end GPUs for compute servers available on the market. Their sole purpose is to be used as a computing unit - since they often don't allow to (directly) connect a monitor or any other visual device to it. The massively parallel GPU architecture provides several times higher theoretical memory bandwidth and floating point performance than CPUs, but are not equally suited for all types of workload. Research has been done on utilizing the GPUs for different types of scientific computations with many successful results [BG08] [ZDG+13] [Gli13], but there are many challenges to utilizing the GPU efficiently [EBW11] [VCC+10].

In this work, our focus was to investigate the application of GPUs for solving dynamical optimization problems, such as optimization problems resulting from model predictive control. The solution of these problems can be subject to hard real-time constraints, which demand fast solution. To solve these problems, often either a simplex or interior-point method is used.

We have focused on utilizing a GPU for accelerating interior point methods, and we have implemented a primal-dual interior point method. The main computa-

tional task in interior point methods is the computation of the Newton direction in each iteration. We have investigated and presented multiple approaches to computing the Newton direction using a GPU, including direct solvers using Cholesky factorization and iterative solvers using preconditioned conjugate gradient.

Based on an existing high-performance library, which includes dense Cholesky factorization on the GPU [ICL], we have presented an implementation of a primal-dual interior point method. The implementation uses a GPU to efficiently solve a model predictive control problem, by using the GPU to construct and factorize the normal equations matrix of the KKT conditions. This implementation was shown to be substantially faster than an equivalent CPU implementation for our test problem, which had a near-dense normal equations matrix. The use of dense factorization limits the problem size to problems where the dense factorization can reside in memory, however, for inherently dense problems the GPU can provide substantial reduction in the solution time. By exploiting the structure of the test problem as described in [ESJ09], we reduced both the solution time and the memory usage. Thus, it was possible to solve larger systems, but the problem size is still very limited by the dense factorization.

For sparse problems, the use of dense factorization is not an optimal choice. Sparse Cholesky factorization, which utilizes sparsity preserving orderings, can significantly reduce the memory usage and computational load. The dense normal equations matrix in our test problem was caused by near-dense rows in the constraints matrix. By formulating the test problem in the standard form instead of the inequality form, we obtained a larger, but sparse, optimization problem. Using an existing sparse Cholesky factorization package, CHOLMOD [Dav], we implemented a solver to compute the Newton direction in our interior point method. This made it possible to solve much larger systems than the dense implementation, and the solution time was also lower, despite the larger number of decision variables in the optimization problem. CHOLMOD is originally a CPU-based implementation, however recent versions have included GPU acceleration for NVIDIA GPUs. Unfortunately, we were not able to make this work for our tests, as the GPU-enabled library resulted in segmentation faults within the CHOLMOD library. A new beta version of CHOLMOD was recently released, which may resolve this issue, but we have not been able to run new tests to include in this thesis. Sparse Cholesky factorization with GPU acceleration has been investigated by multiple authors [GSG+11] [ZD12] [ZDG+13] with results showing a potential two to four times speed-up by utilizing a GPU. We believe this should be investigated further in the future.

An alternative to direct solvers is iterative methods. As the interior point method approaches the solution, the normal equations matrix becomes

ill-conditioned due to the partition displayed by the complementarity pairs [Wri97]. While direct solvers are generally unaffected by this [Gon12b], iterative methods are highly susceptible to ill-conditioning and fails to converge. In [Gon12b], Gondzio presents an interior point method which uses conjugate gradient with regularization and a preconditioner. We have implemented the proposed method for a GPU, and evaluated its performance. The main computational work in conjugate gradient is matrix-vector multiplication, which the GPU is well-suited for [Sø12] [BG08] [BG09], and which makes this an attractive method for GPU implementation. While the conjugate gradient solver fails to converge to the same accuracy as the direct method due to the ill-conditioning, the regularization and preconditioner makes it feasible to use conjugate gradient to compute an approxiate solution. Since there is no factorization stored, it is possible to solve even large systems of the dense formulation. This gave substantial speed-up, compared to the direct method for our test problem.

The different methods we have implemented have been combined into a single modular toolbox called GPUOPT. GPUOPT contains our implementation of a primal-dual interior point algorithm for linear optimization problems based on Mehrotra's predictor-corrector method [Meh92], and also features multiple centrality correctors [Gon96], and weighted corrector directions [CG08]. It is implemented, such that all operations can execute entirely on the GPU. The interior-point method in the toolbox is split into modules, such that the matrix operations and Newton solver is separated from the core interior point method. This allows for problem-specific implementation of the operations, which was shown to reduce the solution time substantially in Chapter 4. The different approaches to the solution of the Newton direction have been implemented as modules for GPUOPT, making them easy to use. GPUOPT is released as open-source as part of this thesis.

Overall, the use of a GPU for interior point methods has shown promising results. For dense problems, which fit within memory, the dense factorization on the GPU performs substantially better then the CPU, due to the high memory bandwidth of the GPU. For larger problems which do not fit within memory, the iterative solver based on conjugate gradient can compute an approximate solution. The performance of conjugate gradient is dependent on the matrix-vector multiplication, which is a memory-bound operation. By utilizing the high memory bandwidth of the GPU, this can be accelerated substantially. To fully utilize the GPU, it is necessary for the optimization problem to contain some dense substructures, such as our test problem, which contains a dense block-Toeplitz matrix. For very sparse problems, the CPU remains competitive with sparse Cholesky factorization. Although not done in this thesis, research has shown that the GPU is also capable of accelerating this operation [GSG+11, ZD12, ZDG+13].

7.1 Future perspective

With this work, we have tried to cover different approaches to using a GPU to accelerate interior point methods and develop a toolbox, which provides a basis for GPU-accelerated interior point methods. There are still many improvements which can be applied to the work carried out in this thesis. Below we summarize some of the future improvements we would like to see, which can help direct future research in the area.

Quadratic optimization problems are currently not supported by GPUOPT, as we restricted ourselves during the thesis to linear optimization problems. However, with minor modifications, the toolbox can be extended to handle quadratic optimization problems. We believe that the GPU can be competitive for dense QP problems.

GPU-accelerated sparse Cholesky factorization has shown to provide speed-up in [GSG+11, ZD12, ZDG+13]. Whether this speed-up also applies to problems from model predictive control remains to be seen, but warrants further investigation.

OpenCL has matured substantially over the last 2-3 years. While CUDA remains the leader in ease-of-use with mature libraries available, multiple OpenCL libraries have been released, and are rapidly catching up. It would be interesting to see GPUOPT implemented with OpenCL instead of CUDA, which would allow it to run on more than just NVIDIA GPUs, such as Intel MIC processors and AMD graphics cards.

Appendix A

Published papers

The following papers were published during the period of this Ph.D project.

1. John Bagterp Jørgensen, Gianluca Frison, Nicolai Fog Gade-Nielsen, and Bernd Dammann. Numerical Methods for Solution of the Extended Linear Quadratic Control Problem, volume 4 of *IFAC Proceedings Volumes (IFAC-PapersOnline)*, pages 187-193. International Federation of Automatic Control, 2012.

2. Søren Schmidt, Nicolai Fog Gade-Nielsen, Martin Høstergaard, Bernd Dammann, and Ivan G. Kazantsev. High resolution orientation distribution function. *Materials Science Forum*, volume 702-703, pages 536–539, 2012.

3. Nicolai Fog Gade-Nielsen, John Bagterp Jørgensen, and Bernd Dammann. MPC Toolbox with GPU Accelerated Optimization Algorithms. *The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012)*. Technical University of Denmark, 2012.

Bibliography

[Acc12] AccelerEyes. Jacket discontinued. http://blog.accelereyes.com/blog/2012/12/12/exciting-updates-from-accelereyes/, 2012.

[AG99] Anna Altman and Jacek Gondzio. Regularized symmetric indefinite systems in interior point methods for linear and quadratic optimization. *Optimization Methods and Software*, 11(1-4):275–302, 1999.

[AGMX96] E. Andersen, J. Gondzio, C. M'esz'aros, and X. Xu. Implementation of interior-point methods for large-scale linear programming, 1996.

[BFV92] John R Birge, Robert M Freund, and Robert Vanderbei. Prior reduced fill-in in solving equations in interior point algorithms. *Operations Research Letters*, 11(4):195 – 198, 1992.

[BG08] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[BPM10] Jakob Bieling, Patrick Peschlow, and Peter Martini. An efficient GPU implementation of the revised simplex method. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[CB04] E.F. Camacho and C. Bordons. *Model Predictive Control*. Advanced Textbooks in Control and Signal Processing. Springer London, 2004.

[CG08] Marco Colombo and Jacek Gondzio. Further development of multiple centrality correctors for interior point methods. *Computational Optimization and Applications*, 41(3):277–305, 2008.

[CN07] Frank Curtis and Jorge Nocedal. Steplength selection in interior-point methods for quadratic programming. *Applied Mathematics Letters*, 20(5):516 – 523, 2007.

[CNW09] Frank E. Curtis, Jorge Nocedal, and Andreas Wächter. A matrix-free algorithm for equality constrained optimization problems with rank-deficient jacobians. *SIAM Journal on Optimization*, 20(3):1224–1249, 2009.

[Col07] Marco Colombo. *Advances in interior point methods for large-scale linear programming*. PhD thesis, University of Edinburgh, 2007.

[Dav] Timothy A. Davis. CHOLMOD: supernodal sparse Cholesky factorization and update/downdate. http://www.cise.ufl.edu/research/sparse/cholmod/.

[Dav13] Timothy A. Davis. User Guide for CHOLMOD: a sparse Cholesky factorization and modification package. http://www.cise.ufl.edu/research/sparse/cholmod/CHOLMOD/Doc/UserGuide.pdf, 2013.

[EBW11] Dominic Eschweiler, Daniel Becker, and Felix Wolf. Patterns of inefficient performance behavior in gpu applications. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 262–266, Washington, DC, USA, 2011. IEEE Computer Society.

[Edl10] Kristian Skjoldborg Edlund. *Dynamic Load Balancing of a Power System Portfolio*. PhD thesis, Aalborg Universitet, 2010.

[ESJ09] Kristian Edlund, Leo Emil Sokoler, and John Bagterp Jørgensen. A primal-dual interior-point linear programming algorithm for MPC. In *CDC*, pages 351–356. IEEE, 2009.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 216–225, Washington, DC, USA, 2011. IEEE Computer Society.

[Gli13] Stefan Lemvig Glimberg. *Designing Scientific Software for Heterogeneous Computing*. PhD thesis, Technical University of Denmark, 2013.

[Gon96] Jacek Gondzio. Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications*, 6(2):137–156, 1996.

[Gon12a] Jacek Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[Gon12b] Jacek Gondzio. Matrix-free interior point method. *Computational Optimization and Applications*, 51(2):457–480, 2012.

[GSG+11] T. George, V. Saxena, A. Gupta, A. Singh, and A.R. Choudhury. Multifrontal factorization of sparse SPD matrices on GPUs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 372–383, May 2011.

[Har07] Mark Harris. Optimizing parallel reduction in CUDA. 2007.

[HEBJ10] T.G. Hovgaard, K. Edlund, and J. Bagterp Jorgensen. The potential of Economic MPC for power management. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 7533–7538, Dec 2010.

[HJO07] Jin Hyuk, Jung, and Dianne P. O'leary. Implementing an interior point method for linear programs on a CPU-GPU system, 2007.

[HJPM14] Rasmus Halvgaard, John Bagterp Jørgensen, Niels Kjølstad Poulsen, and Henrik Madsen. *Model Predictive Control for Smart Energy Systems*. PhD thesis, Technical University of Denmark, 2014.

[ICL] The University of Tennessee Innovative Computing Laboratory. MAGMA linear algebra library. http://icl.cs.utk.edu/magma/.

[KmWH10] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, first edition, 2010.

[LBEB11] Mohamed Esseghir Lalami, Vincent Boyer, and Didier El-Baz. Efficient implementation of the simplex method on a CPU-GPU system. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1999–2006, Washington, DC, USA, 2011. IEEE Computer Society.

[LTND10] Hatem Ltaief, Stanimire Tomov, Rajib Nath, and Jack Dongarra. Hybrid multicore cholesky factorization with multiple GPU accelerators, 2010.

[Mac02] J.M. Maciejowski. *Predictive Control with Constraints*. Pearson Education. Prentice Hall, 2002.

[MAC11] Xavier Meyer, Paul Albuquerque, and Bastien Chopard. A multi-*gpu* implementation and performance model for the standard simplex method. 2011.

[Meh92] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. volume 2, page 575–601, 1992.

[NIS] NIST. Matrix Market. http://math.nist.gov/MatrixMarket/.

[NVIa] NVIDIA. CUBLAS - the NVIDIA CUDA basic linear algebra subroutines. http://developer.nvidia.com/cuBLAS.

[NVIb] NVIDIA. CUSPARSE - the NVIDIA CUDA sparse matrix library. http://developer.nvidia.com/cuSPARSE.

[NVIc] NVIDIA. GPU accelerated libraries. http://developer.nvidia.com/gpu-accelerated-libraries.

[NVId] NVIDIA. Thrust. http://developer.nvidia.com/thrust.

[NVI09] NVIDIA. Fermi compute architecture whitepaper, 2009.

[NVI12] NVIDIA. Kepler GK110 architecture whitepaper, 2012.

[NVI13a] NVIDIA. CUDA C best practices guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2013.

[NVI13b] NVIDIA. CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2013.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, 2. ed. edition, 2006.

[RM09] J.B. Rawlings and D.Q. Mayne. *Model Predictive Control: Theory and Design*. Nob Hill Publishing, 2009.

[SE09] Daniele G. Spampinato and Anne C. Elster. Linear optimization on modern GPUs. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–8. IEEE, 2009.

[SGH12] Edmund Smith, Jacek Gondzio, and Julian Hall. GPU acceleration of the matrix-free interior point method. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 681–689, Berlin, Heidelberg, 2012. Springer-Verlag.

[She94] Jonathan R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. August 1994.

[Sø] Hans Henrik Brandenborg Sørensen. GLAS download. http://gpulab.imm.dtu.dk/software.html.

[Sø12] Hans Henrik Brandenborg Sørensen. Auto-tuning dense vector and matrix-vector operations for Fermi GPUs. In *Parallel Processing and Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science*, pages 619–629. Springer Berlin Heidelberg, 2012.

[VCC+10] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[Wri97] S.J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997.

[ZD12] Dan Zou and Yong Dou. Implementation of parallel sparse cholesky factorization on gpu. In *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*, pages 2228–2232, Dec 2012.

[ZDG+13] Dan Zou, Yong Dou, Song Guo, Rongchun Li, and Lin Deng. Supernodal sparse cholesky factorization on graphics processing units. *Concurrency and Computation: Practice and Experience*, 2013.